Empirical research using R:
in economics, finance, and management
Essentials, real examples, and troubleshooting

Compiled from session01.tex @ 2024-06-21 15:33:13+02:00.

Day 1: Introduction into programming

Andreï V. Kostyrka

18th of September 2023

UNIVERSITÉ DU LUXEMBOURG

Presentation structure

1. Administrative formalities
2. Comparison of statistical packages
3. How computers compute and store data
4. Programming concepts
5. R as a programming language

Administrative formalities

Why this course exists

- Quantitative methods in economics have never been more popular
- There are many fragmented online tutorials and Q&A's (oversimplified to the point of being unreliable)
- Last R course at Uni.LU: 2018 (by Dr. Laurent Bergé)
  - Inspired me to switch from fragmented scripts to writing structured packages
  - Main difference: LB focuses on package development and low-level building blocks – this course teaches general problem solving in R for Ph.D.'s
- Promotion of open science and reproducible research at Uni.LU

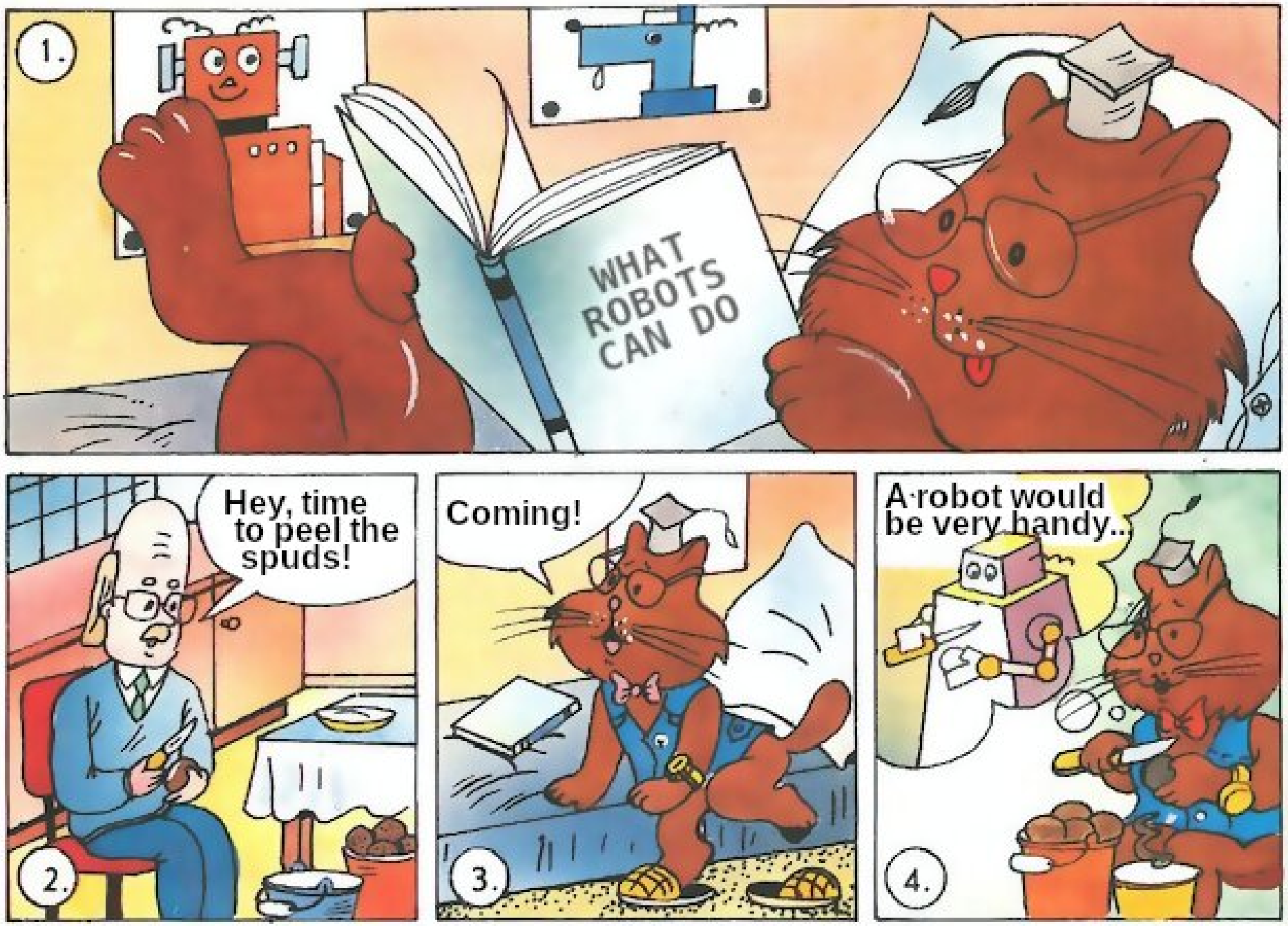
How this course was shaped

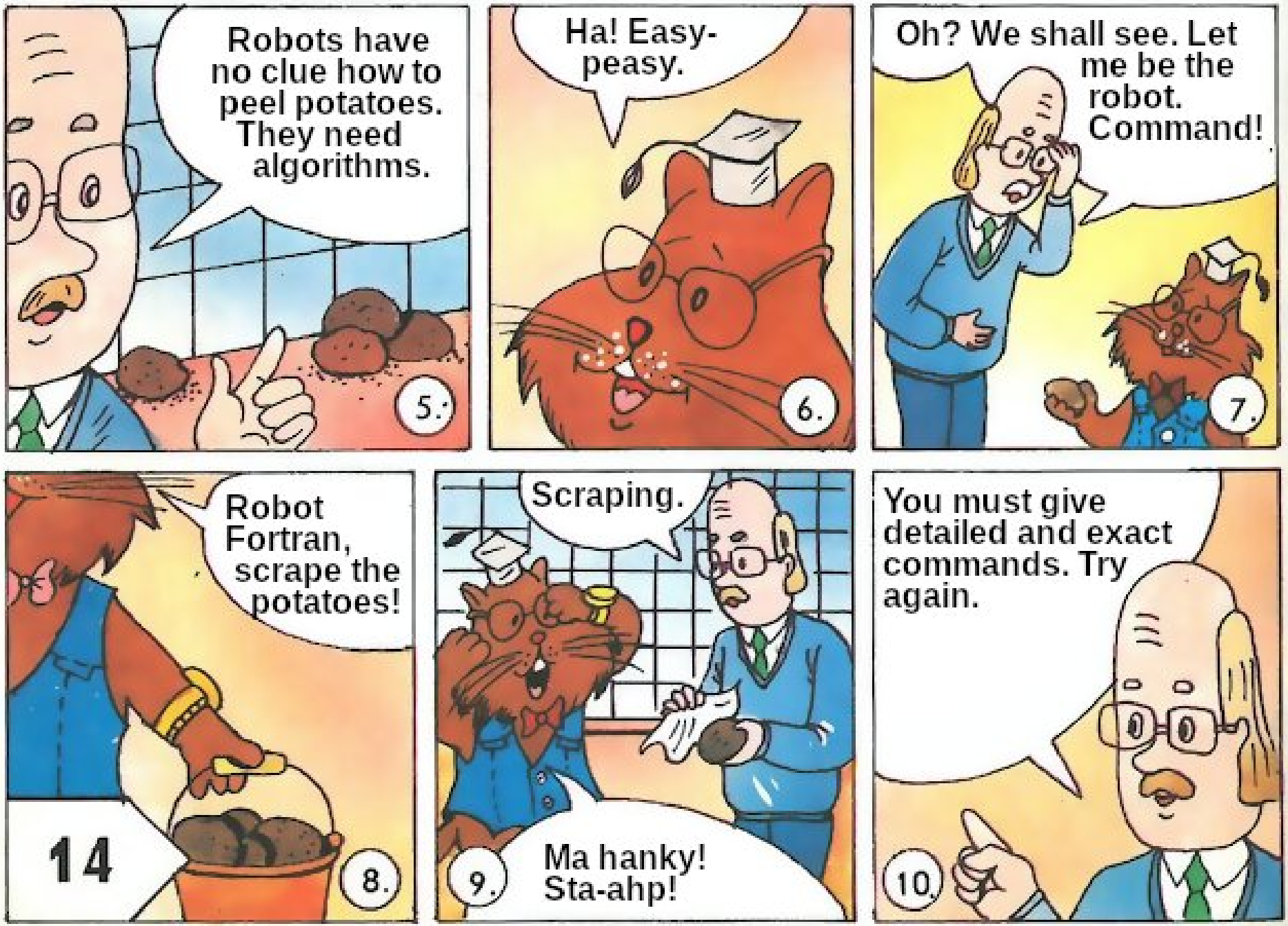
- Over the course of my doctoral studies, I received dozens of questions from fellow students
- Students wanted to know how to
  1. Answer highly specific research questions
  2. Implement certain methods
  3. Check if their implementation was correct
  4. Produce non-standard graphics
- As a result, this course is based on DSEFM Ph.D. students' concrete questions, data sets, and examples

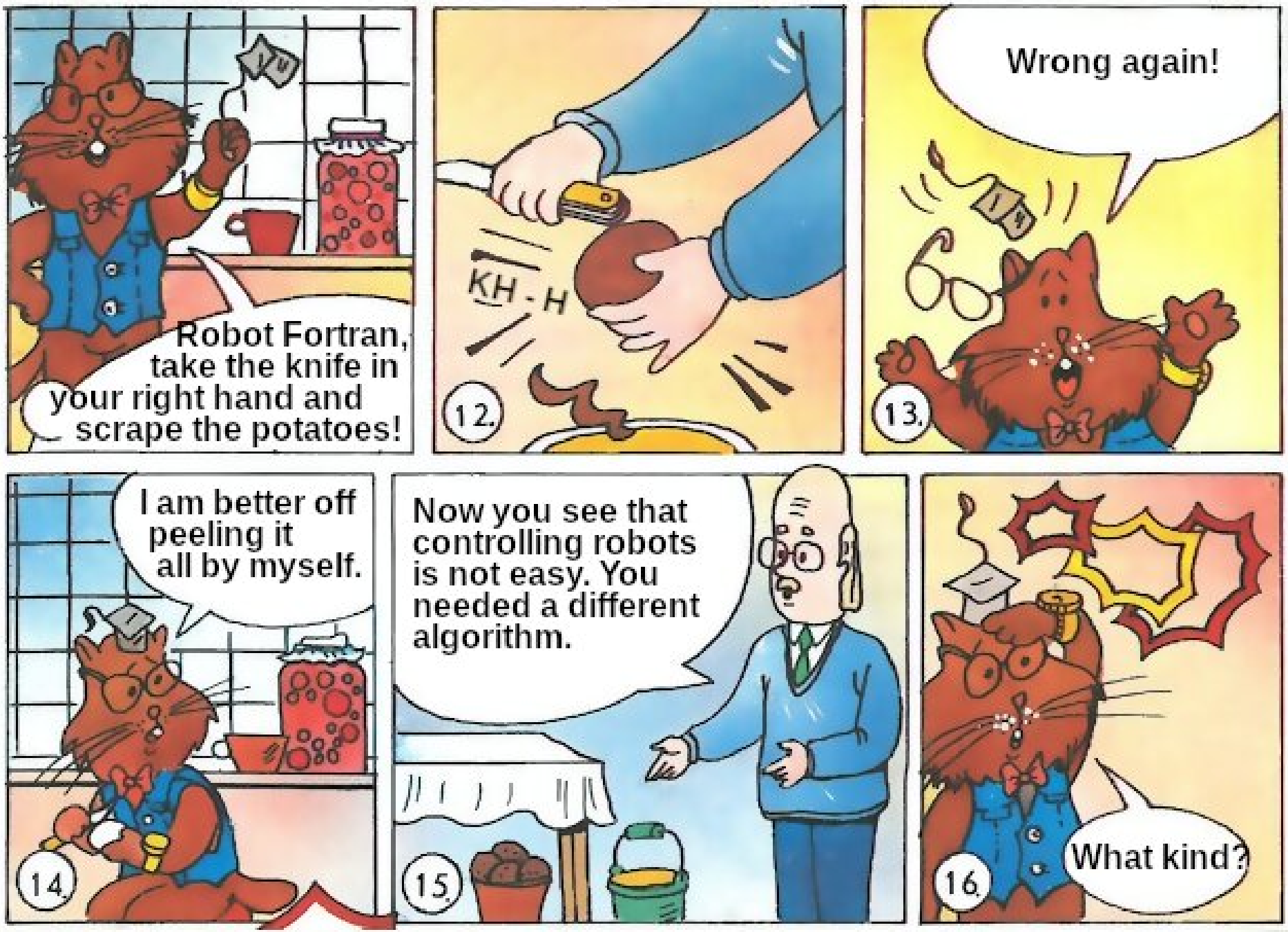
Course goals

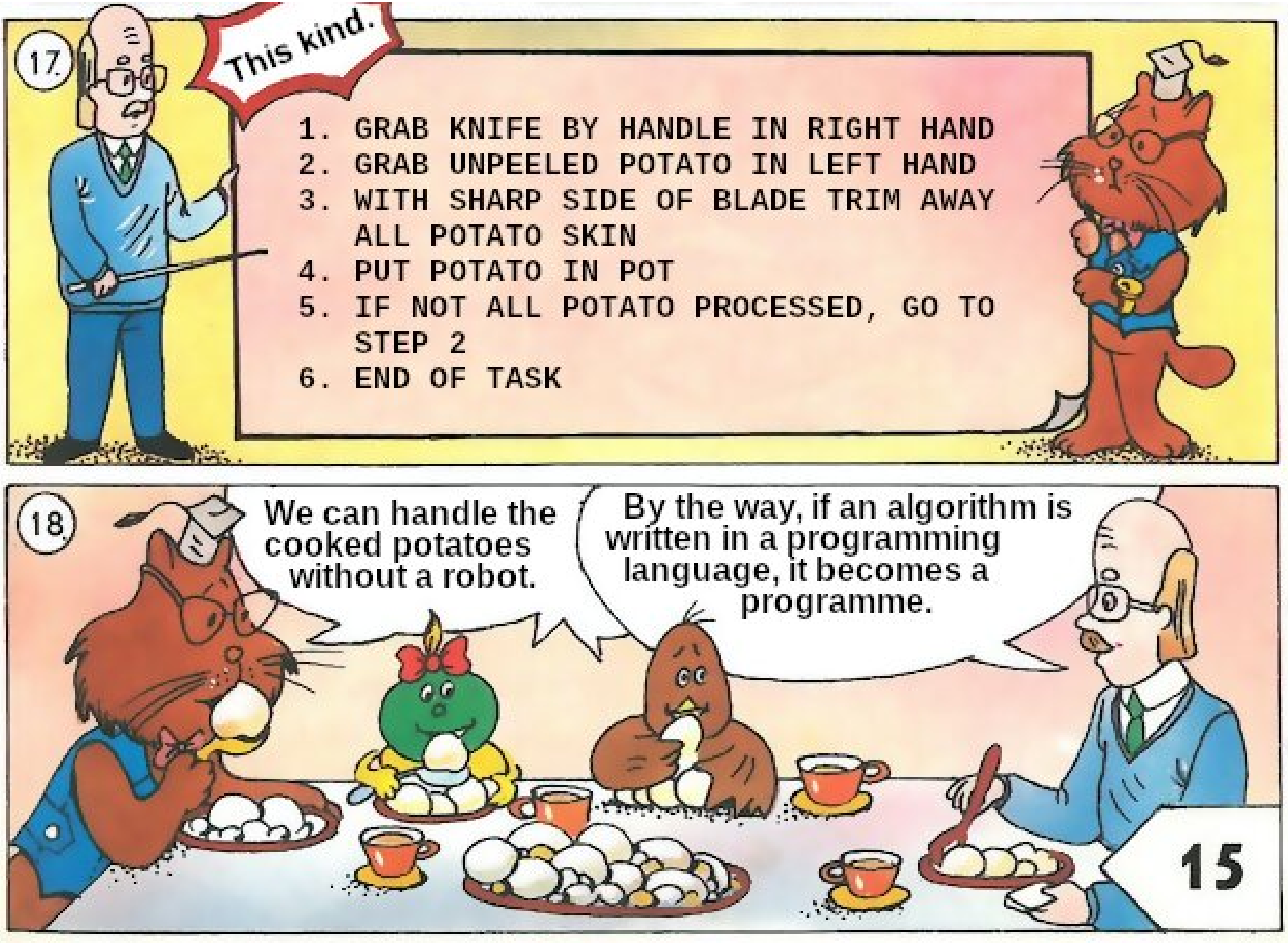
- Help researchers learn the basics of R and apply it to their research questions
- Teach how to produce useful diagnostics and visualise the results
- Guide how to proceed in case of troubles or errors
- Teach a bit of programming culture and *algorithmic thinking* in general to break down complex problems into computable steps

Professor Fortran and his friends will illustrate the point.


1.
WHAT ROBOTS CAN DO
Hey, time to peel the spuds!
2.
Coming!
3.
A robot would be very handy...
4.


Robots have no clue how to peel potatoes. They need algorithms.
5.
Ha! Easy-peasy.
6.
Oh? We shall see. Let me be the robot. Command!
7.
Robot Fortran, scrape the potatoes!
8.
14
Scraping.
9.
Ma hanky! Sta-ahp!
You must give detailed and exact commands. Try again.
10.


Robot Fortran, take the knife in your right hand and scrape the potatoes!
KH-H
12.
Wrong again!
13.
I am better off peeling it all by myself.
14.
Now you see that controlling robots is not easy. You needed a different algorithm.
15.
What kind?
16.



Credit: 'Professor Fortran's Encyclopaedia' (1991).

Why R?

- It is free
- It is a programming language (≠ Stata): coherent & smart syntax, loops and conditions (one rarely needs more)
  - Interpreted, object-oriented, functional
  - Easier and more intuitive than hard-core programming languages
- It has a very smart development environment with very helpful interface features (RStudio)

Competitive advantages

- Many built-in one-liners for popular methods
  - 20,000 packages for data cleaning, wrangling, visualisation, estimation, and more
  - Easy to write functions to automate routines
- Vectorised operations, easy parallel computations with big data
- Handles as many objects (data bases, models, parameter sets etc.) as one desires
- High-quality plots (even 3D, even animations!)

R and yours truly

- Discovered R in 2015 (after a long affair with gretl) – since then, a true love story
- Helped me not only in Ph.D., but in real life (as a Swiss knife)
- Daily coding since then (and the best pocket calculator!)

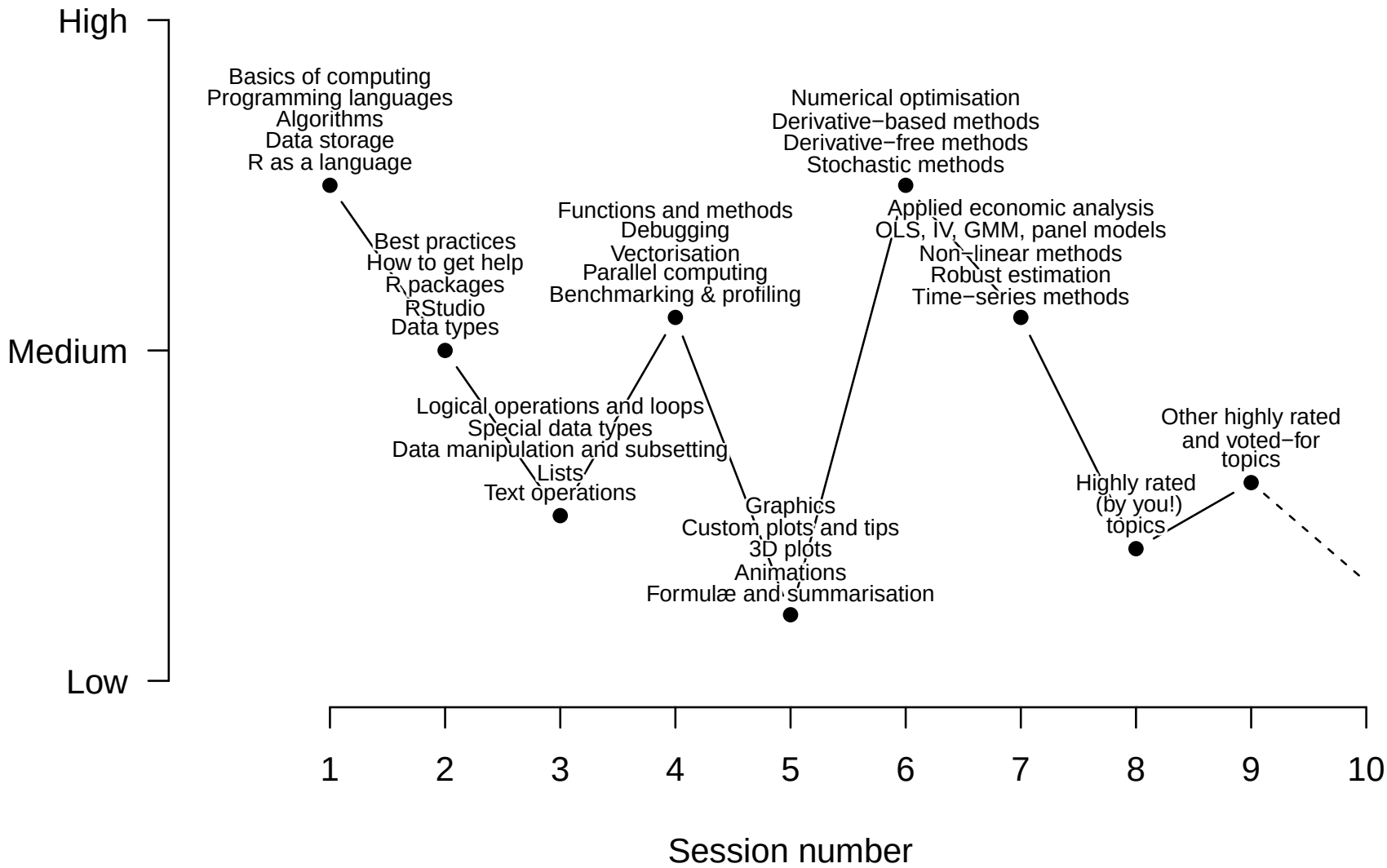
What this course is not

- Not about narrow or highly specialised packages
  - There are too many narrow tutorials on R packages for data cleaning, wrangling, exploratory analysis etc.
  - Bias in the R literature in favour of specific black-box solutions from this-and-that package
- Not econometrics or statistics (a good thing!)
  - Ask questions on those subjects afterwards, and I shall provide references
- Not computer science or algorithm analysis

What this course is

- Basic understanding of the core R syntax
- Writing functions to solve a **broad spectrum of problems arising in applied economics**
  - Self-help: Handling errors, debugging functions, getting help, troubleshooting economic models
- Creating plots of arbitrary complexity step by step
- Showcasing real-world economic applications of highly customisable functions

Amount of technical detail per session



During the course

- 10-minute break in the middle
- The schedule is intense; study at home, write down questions about what is unclear, ask them during the sessions
- Having a laptop is completely optional (you can follow the screen), but running the code at least once in your spare time is a must

The syllabus

- Contains the intended agenda
- Contains links to openly published learning resources (books, online tutorials etc.)
  - Your suggestions of learning resources are welcome
- Contains howework descriptions and full final project proposals

Grading

- 10% attendance + 3 · 15% assignment + 45% project
- The assignments are short and must be done to reinforce learning
- Final project: **choose the task that is the most relevant for your research** or the one that can be later reused in other projects
  - You may reuse your existing material in the assignment

Technical requirements for all assignments

1. The implementation must be done in **base R + 'CRAN recommended packages'**
   - Do not load random libraries; learn to create tools
2. If your model is hard to solve or simulate, you are allowed to load a dedicated package for working with such models
3. The following packages **must not be used:** `ggplot2`, `dplyr`, `tidyr`, `purrr`, `tibble`, `stringr`, `forcats`, `magrittr`, `tidymodels`
4. `data.table` may be used only if a benchmark shows a >50% faster run time or if built-in functions halt

Any questions on the formalities?

Comparison of statistical packages

What to expect from a software package

- Should be free or affordable
- Should be extensible
- Should not hold the user hostage
- Should have the means to focus on research without low-level quirks and allow the user to **write good algorithms**

R is not just base R

R is an ecosystem similar to Python, Perl, Ruby, TEX etc.

- Basic installation: binary executable (providing the **console**) + core libraries (graphics, stats, utilities etc.)
- User interface: IDE
  - RStudio is the industry standard, suits most users
  - Visual Studio Code plugins, editors with just syntax highlighting etc.
- User packages: pure R (interpreted) or compiled (C++ via RCpp, Fortran etc.)
  - Might need dependencies (e.g. `curl` for fetching data)
  - Can be installed / updated from within R

Stata

- Software package for script-based estimation (GUI exists, but incomplete) + matrix manipulation language
- Created in 1985
- Comes in various editions (only difference: parallel support + data size limitation)
- Popular among applied economists
  - Has a dedicated journal ('Stata journal') since 2001

Stata strong points

- Easy to use for popular types of analysis
- Has almost all cookbook methods with some degree of flexibility out of the box
  - Most of them are very intuitive and cover a wide range sufficient for many researchers
- Documentation is clear, detailed, and has good theoretical insights
- Many reliable user-written packages (RePEc BoCode)
- Functions in ADO files are virtually open-source
  - Anybody can look into the implementation of any invokable function

Stata shortcomings

- Not flexible for Ph.D. in economics as the only tool
  - Data transformation is a huge pain
  - Few recent methods, new functions hard to implement
- Very expensive (hundreds of €)
  - Violates the principles of open science
- Confusing interaction with environment objects, severely limited interactivity, hard to debug
  - Number? `display`. Vector? `matrix list`!
- User interface gives access to a mere fraction of command parameters
- In the past: 32-bit storage format (23 significant bits) macheps $\approx 10^{-7}$ – heavily corrupted saved data

EViews

- Software package for script-based procedures
- Created in 1994
- Popular among macroeconomists
- Main focus: time-series analysis and forecasting, systems of dynamic equations, simulations of shocks

EViews strong points

- Excellent support for the variety of basic time-series methods, tests, and method-specific diagnostics and plots
- Out-of-the box support for fetching data from commercial sources (FRED, DBnomics, World Bank, Bloomberg etc.)
- Really easy to define state-space models and manage hundreds of connected equations
- Developers answer paying users' questions on the forum within a couple of days

EViews shortcomings (my pulse went up)

- Almost impossible to develop custom tools
  - Common mathematical / statistical functions missing
  - Dated / limited / incomplete out-of-the box methods
- Unstable, obsolete, uncustomisable numerical solvers
  - No constrained optimisation, strange stopping criteria
- Some functions have hidden bugs (e. g. LASSO)
  - No way for the user to fix those bugs
- Data manipulation methods even poorer than in Stata
- Zero interactivity / diagnostic methods (black box)
- Hurts and frustrates the user in many ways
  - Code editor had no dark scheme before my 2022 request
  - No global variable support
- Has no learning resources sans the forum and manual

SPSS

- Software suite with emphasis on GUI (command syntax language support exists)
- Created in 1968
- Various editions exist (base, standard, premium etc.)
  - Has purchasable addons (e. g. Amos for structural equation modelling)
- Very popular among sociologists and psychologists

SPSS strong points

- Easy to use for popular types of analysis
  - Allows researchers without any knowledge of statistics or coding skills to conduct analysis in several mouse clicks
- Intuitive user interface with convenient navigation
- Easy to estimate models with features common in sociology / psychology (tricky linear contrasts, partial interaction contrasts etc.)

SPSS shortcomings

- Ridiculously expensive (3600+ $)
  - Little community support, no StackExchange, no strong online culture with enthusiastic users
- Lacks some basic functionality (e. g. robust standard errors or non-parametric regression!)
- Insurmountably hard to implement a custom model
  - State-of-the-art models are not added for several years
- Weak facilities for organising a workflow with pipelines
  - Or choosing between multiple models automatically depending on the fit / forecast quality
- SPSS UI discourages coding, which hampers quality assurance, process review, and replication
  - Hayes (2022): 'Users of SAS and R **have no choice but** to write in code' (as if it were a bad thing!)

SAS

- Software suite with emphasis on command-line syntax
- Created in 1972
- Has over 200 modules for various tasks
  - Has a cloud-based soluton, Viya
- General programming language with emphasis on database handling, specialised analytics, and visualisations

SAS strong points

- Loads big data sets very quickly without overloading RAM
  - Could handle big data even 50 years ago
- Comprehensive logging for troubleshooting
- Particular applications: complex error structures in mixed-effects models (and fast)

SAS shortcomings

- Prohibitively expensive (8000+ $ – even American Express dropped it!)
  - No community support, no StackExchange
- Not usable on Mac & Linux (only VMs)
- Memorisation rather than conceptual understanding
  - Idiosyncratic, not useful for other programming languages
- Historical ballast: one thing / step, proc this proc that
  - Arduous data set manipulation not for the modern age
- Does not return objects to further manipulate
- Macros contaminate the global environment or rely on global variables

Python

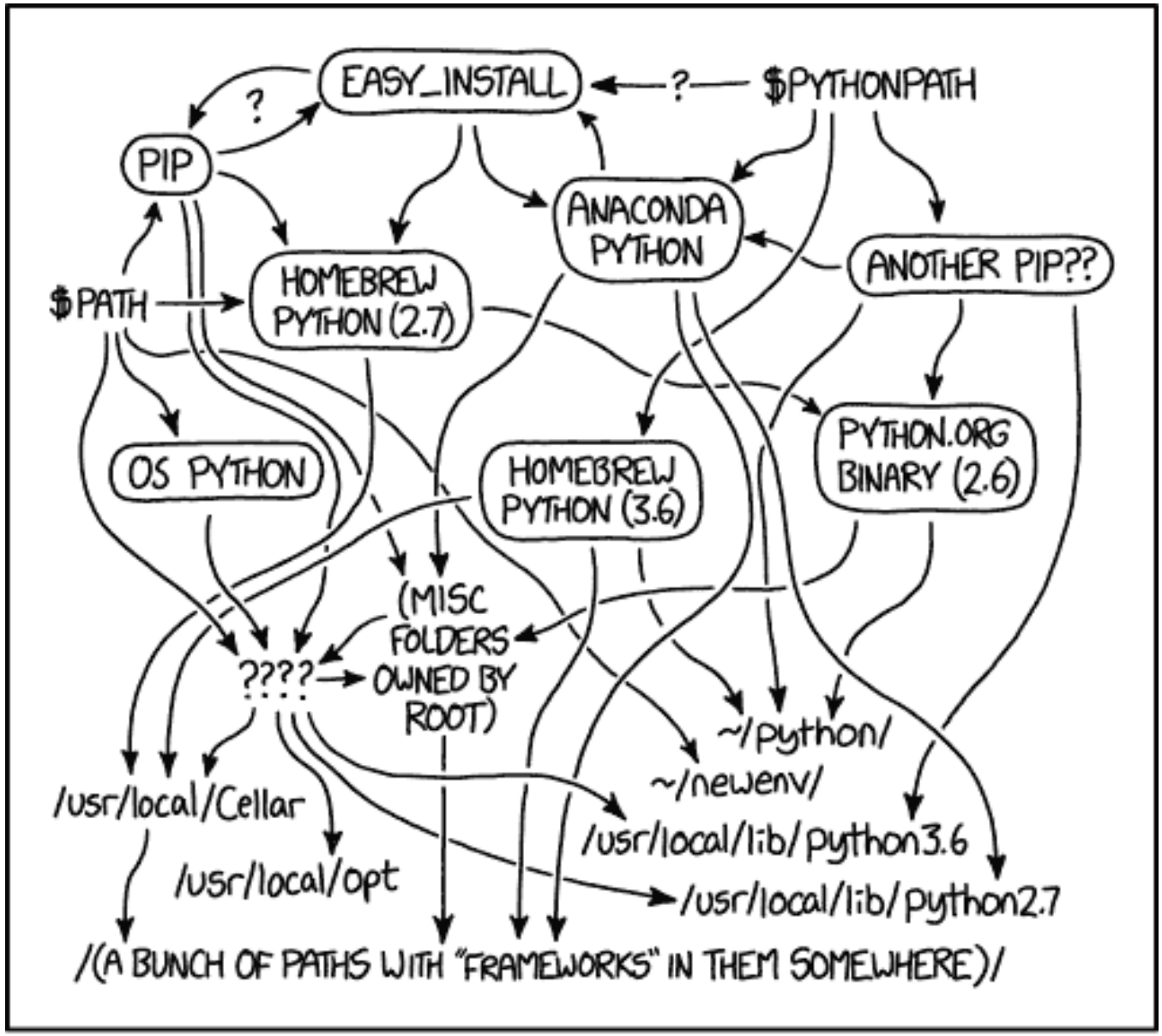
- High-level general-purpose programming language
- Created in 1991
- Designed to be extensible via modules, has over 200,000 official packages
- Has many wrappers around C/C++ libraries, e. g. PyTorch
- Very popular among CS researchers and people in the machine-learning and artificial-intelligence industry
- **The** most popular programming language in the world since October 2021 (as of September 2023)

Python strong points

- Free and supported on all platforms that can run DooM
- Easy to get quick results with little programming skill
- Machine-learning capabilities make it the highest-ranking language in prediction competitions (e. g. Kaggle, M5 etc.)
- Best for prototyping in many areas of research and for machine learning
- Looks very good on one's CV to Big Tech companies
- Libraries for virtually all aspects of computing, not only economics
- Good enough for many practical applications
  - Teaching algorithms and robotics to kids

Python shortcomings

- No dedicated focus on research in economics
  - Joke (Michael Reeves): 'Python can do anything, just badly'
- Slow, bad as the sole language for large-scale projects
  - Common to start in Python and rewrite in C or Rust
- Dependency hell: many projects depend on specific versions
  - Multiple projects, multiple environments and packages
- Code rot: evolves so quickly, hard to reproduce research after mere months
  - Transition from Python 2 to Python 3 was painful
- *For economists:* Many statistical methods are implemented by CS researchers, not statisticians
  - Can be poorly documented or strangely implemented



MY PYTHON ENVIRONMENT HAS BECOME SO DEGRADED
THAT MY LAPTOP HAS BEEN DECLARED A SUPERFUND SITE.

Credit: xkcd. Technical explanation.

R

- High-level statistical programming language
- Created in 1993
- Extensible via user packages, has over 20,000 in the official repository (CRAN)
- Written in Fortran and C ⇒ the base is designed to be as fast as possible
- Very popular among researchers in bioinformatics and genomics, data miners
- #16 most popular programming language (2023-05; was #8 in 2020-08) (the only statistics-oriented one in the top 20; MATLAB is #15)
  - Has a dedicated journal ('R journal') since 2009

R strong points

- Free, open-source, extensible, feature-rich
  - Huge active community, many answers to users' questions
  - CRAN package maintainers are responsible for downstream compatibility; version / package / environment issues are much rarer than in Python
- One can easily code whatever they want in multiple possible ways (great for highly custom research)
- Has some very smart and convenient coding features
- There are many extensions: Markdown & Shiny
  `shiny.rstudio.com/gallery/superzip-example.html`
  `gallery.shinyapps.io/shiny-salesman`
- Possible to create one's own packages

R shortcomings

- Slightly steeper learning curve, especially for users without command-line experience
  - Lots of tutorials ⇒ lots of **bad** tutorials
- Was not created by a coordinated team ⇒ invocation of certain features can be clunky (e. g. parallelism)
  - Function naming principles are sometimes inconsistent
  - Multiple functions may implement the same concept (e. g. `prcomp()` and `princomp()` with different structure)
  - Syntax depends on the contributor of that function
- Quality control depends on the programmer if external packages are used
  - Occasionally, packages are deleted from CRAN – requires an extra step to install the archived version
- Cannot replace all statistical packages with all features

Free & open-source vs. corporate

- Unfair comparison: hard to both commercialise a language and attract enough contributors to allow it to develop according to users' needs
- Programming languages' high flexibility ('real programming') – corporate products are domain-specific (restricted by the logic of their application)
- For economists: Research relying on 30-year-old versions of closed products sold at 8000 $ raises suspicion (pay-to-reproduce)
- Sanity check: search 'why you should abandon SPSS for R' and the opposite, compare the search results

A curious tendency

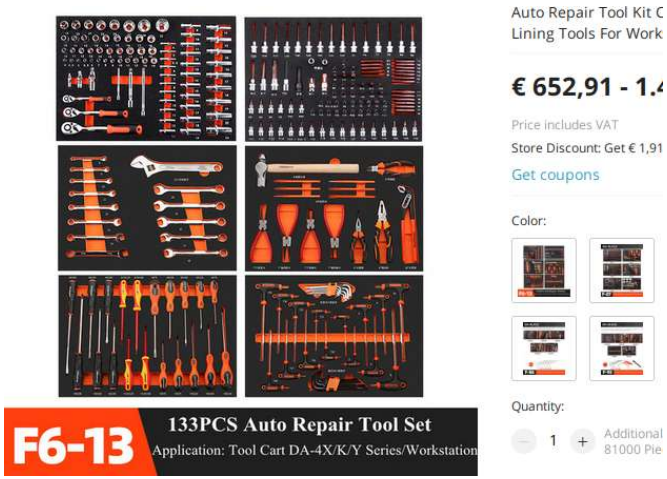
Proprietary software developers are developing ways to integrate their products with open-source programming languages.

- Stata: embed Python or call Stata from Python
- Eviews: R connector
- SPSS: gave up on non-parametric methods, offers an R plug-in
- SAS: Python, R, Java, Lua modules (both free and paid)


SPSS, SAS,
STATA, EVIEWS
OPEN-SOURCE
COMMUNITY
Look what they need to
mimic a fraction of our power
imgflip.com


Are you two friends?
STATA USERS
R USERS
Yes.
No.
imgflip.com

Python vs. R



Python: more versatile, general-purpose; focus on *prediction* via data science, big data, machine learning; written mostly by CS specialists

R: more specalised, more powerful at statistics, bioinformatics, econometrics, visualisations for academia; focus on *estimation*; written mostly by researchers

Software and ‘data science’

- Some words look attractive on a CV; *Python*, *R*, and *data science* are among them
  - Yet one should not glorify tools and say ‘data science’ for ‘using data to solve tasks in a pragmatic manner’
- The tools are there to help one solve problems (research, industry, home projects)
  - R and Python are especially good within a good system of other highly specialised tools that do one task very well

YouTube video by Baba Brinkman: rap battle between a data scientist and a classical statistician, arguing for predictive algorithmic models versus inferential data models respectively

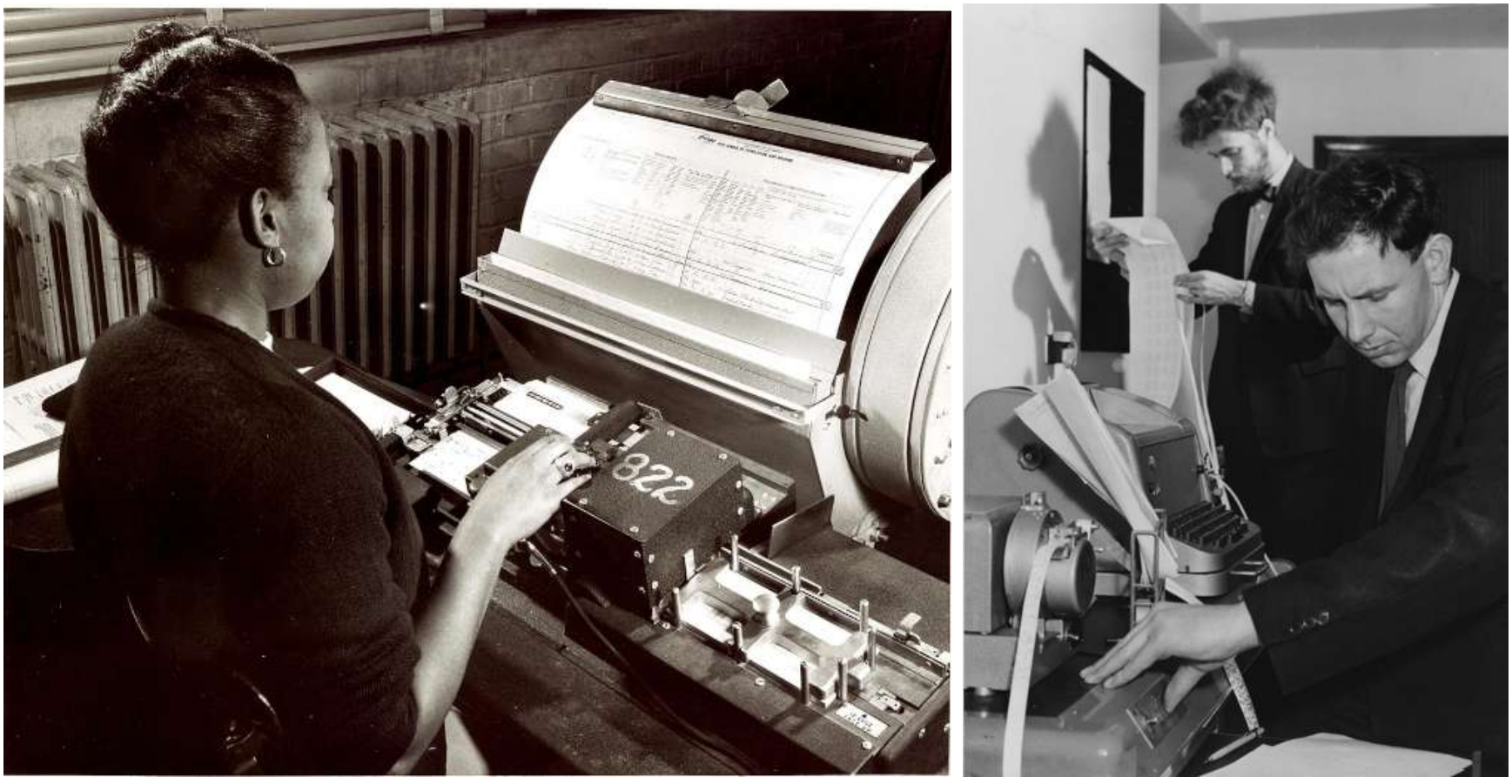
How computers compute and store data

Why do economists use computers?

- The 1880 U.S. census took 8 years to process
- Herman Hollerith used punch cards to speed up the process in 1890
- Since the 1910s, punch-card-based machines were manufactured for accounting
- In the 1940s, the need for computational methods and pseudo-random numbers grew (simulations, Markov chains)
  - The machines used during WWII to decipher German codes were freed

Hollerith’s tabulating machine

Credit: Adam Schuster.



Left: clerk punching a card for the 1950 U.S. census.
Right: students in the machine room of LSE (1964).

Credit: U.S. Census Bureau; LSE.

Types of data we would like to store

- Numbers (integers, reals, decimals)
- Text
- Images, sound, video, 3D models etc.

Types of data computers can store

- Two numbers: **1** and **0**
- We need to construct all other types of data from this

How computers store data

- By default, 8 bits (0 or 1) are grouped into something called a **byte**
- Bytes may be interpreted as numbers or characters
- ASCII: encoding to match numbers to symbols (1 integer in [0, 255] = 1 byte = 1 symbol)
- Unicode (most popular: UTF-8): encoding with more bytes per character to represent more characters

How to read binary

Suppose that we have a byte, i. e. 8 bits:

$$01100101$$

This is how it should be read – each digit corresponding to a power of 2 in base-2 arithmetic:

$$\begin{array}{cccccccc} 0 & 1 & 1 & 0 & 0 & 1 & 0 & 1 \\ 2^7 & 2^6 & 2^5 & 2^4 & 2^3 & 2^2 & 2^1 & 2^0 \end{array}$$

Result:

$$0 \cdot 2^7 + 1 \cdot 2^6 + 1 \cdot 2^5 + 0 \cdot 2^4 + 0 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0$$

$$64 + 32 + 4 + 1 = 101$$

We write it as follows: $101_{10} = 01100101_2$. **Q:** What is 101_2?

How to read hexadecimal (base 16)

Another popular number base is 16 because bytes consist of 8 bits, and those are easy to write with two base-16 numbers.

For base-10 numbers, 10 digits are required (0, 1, 2, 3, ..., 9). For hex numbers, 16 digits are required. The digits larger than 9 are denoted as **a** (10), **b** (11), **c** (12), **d** (13), **e** (14), **f** (15).

Converting $beef_{16}$ to decimal:

$$beef = \underset{11}{b} \cdot 16^3 + \underset{14}{e} \cdot 16^2 + \underset{14}{e} \cdot 16^1 + \underset{15}{f} \cdot 16^0 = 48\,879$$

Q: convert 80085_{10} to hex and 80085_{16} to dec.

Scientific (exponential) notation

$80\,000 = 8 \cdot 10^4$, $0.000\,002 = 2 \cdot 10^{-6}$.

Normalised SN: $a \times 10^b$, where $1 \leq |a| < 10$; a is the **significand (mantissa)**, 10 is the **base**, b is the exponent.

Easy to round: the population of Luxembourg in 2021 was $640\,064 = 6.400\,64 \cdot 10^5 \approx 6.4 \cdot 10^5$.

Scientific notation also exists for binary:

$$1.1101_2 \cdot 2^{11_2} = (1 + 1/2 + 1/4 + 0 + 1/16) \cdot 8 = 14.5$$

Here, $1.1101_2 = 1.8125_{10}$ is the mantissa, 2 is the base, and $11_3 = 3_{10}$ is the exponent.

Floating-point arithmetic

Computers represent real numbers by approximating them with an integer mantissa and an integer exponent:

$$1.8125_{10} = \underbrace{18\,125}_{\text{integer mantissa}} \cdot \underbrace{10}_{\text{base}}^{\overbrace{-4}^{\text{integer exponent}}}$$

The number 18.125 has the same mantissa and a different exponent (–3). The point separating the whole and the fractional part moves: 1.8125 → 18.125.

Such numbers are called **floating-point numbers**.

Performance measure: FLOPS (FP **o**perations **p**er **s**econd).

Available precision

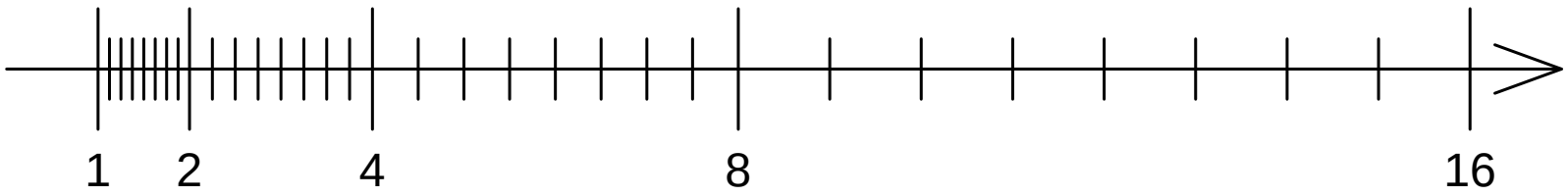
0 10000001101 1100000011010010000000101110100000111010110010111001

- 64 bits are used as follows: 1 for sign, 11 for exponent, 52 for significant digits (IEEE 754 standard)

$$(-1)^{\text{sign}} \cdot (1.\text{significand}) \cdot 2^{\text{exponent}-2^{10}+1}$$

  - In decimal: $\approx 1.7532\ldots \cdot 2^{1037-1023} = 28\,724.5$
- 64-bit FP numbers represent $5 \cdot 10^{-324}$ … $2 \cdot 10^{308}$
- Are 64-bit calculations accurate up to 10^{-323}?
  No, only to $2.2 \cdot 10^{-16} = 1/2^{52}$**!** (Relative accuracy.)
- There are only 52 binary significant digits (≈16 decimal) – after those, the input is truncated
  - Max. relative rounding error is thus 0 … $1.1 \cdot 10^{-16}$
  - Arbitrary precision exists, but is slow

Graphical representation of FP accuracy



- All intervals [1, 2], [2, 4], [4, 8], ...are cut into $2^{52} \approx 4.5 \cdot 10^{15}$ equal intervals. All numbers are rounded to the edge of those intervals
- The gap between two representable numbers is proportional to the number magnitude

Precision loss examples

```
a <- 2^52
b <- a + 0.46
b - a # is equal to what
```

Answer: **zero!**
(Exactly integer zero, no decimal fluff: 0.000...0.)

- The next number after 2^{52} representable by the machine is $2^{52} + 1$ – everything less than $2^{52} + 0.5$ gets rounded down to 2^{52}.

Finite precision with pen and paper

I give you a number: $1/3$. Write it using 4 decimal digits.

- Answer: 0.3333 (rounding error: $-1/30\,000$).

I give you a number: $2/7$. Write it using 3 decimal digits.

- Answer: 0.286 (rounding error: $1/3\,500$).

In our decimal system (base 10), we can represent any integer divided by $2^a \cdot 5^b$ (for integer $0 \leq a, b < \infty$) as a finite decimal fraction.

$1/40 = 1/(2^3 \cdot 5) = 0.025$ – finite.

$1/210 = 1/(2 \cdot 3 \cdot 5 \cdot 7) = 0.0\overline{(047619)}$ – infinite (periodic, but we need to stop somewhere (paper is finite).

Final error scale

- Accurate + inaccurate = inaccurate
- Large abs. err. + Small abs. err. = Large abs. err.
  - Smth big + smth small = as accurate as the big thing

Rule: the harm from the absolute error is minimal when the amounts have the same order of magnitude.

Computations with X_1 = share of population $\in [0, 1]$ and X_2 = GDP (€) $\in [0, 10^{12}]$ are slightly less accurate than with X_2^* = GDP, bn or trn $\in [0, 10]$.

- In research articles: tables with $\hat{\beta}_1 = 12\,325$ and $\hat{\beta}_2 = 0.000\,007\,4$ are annoying!
  - Especially without readability-enhancing thin spaces

Precision loss investigation

- With finite memory per number, all infinite decimal fractions are truncated / rounded to the nearest representable number
- In the following analysis, the sign '$\overset{64}{\scissors}$' shall denote 'is approximated by 64-bit computers through truncation to'

Good decimal, bad binary fractions

Since computers store numbers in binary, only the numbers that can be written as a finite sum of integer powers of 2 are stored losslessly (not powers of 5, alas!).

$1/2 = 0.5_{10} = 0.1_2$ – fine.

$4/5 = 0.8_{10} = 0.1100\,1100\,\ldots_2 = 0.\overline{1100}_2$ – infinite period.

Mantissa is truncated after 52 bits ⇒ we can represent only $0.\underbrace{[1100]}_{\times 12}1100 \overset{64}{\leadsto} 0.8 - 2 \cdot 10^{-16}$ or

$$0.\underbrace{[1100]}_{\times 12}1101 \overset{64}{\leadsto} 0.8 + 4 \cdot 10^{-17}.$$

If 0.8 is saved as a number, it is read back as a **different** one:

```
print(0.8, 20) # 0.80000000000000004441.
```

Computers have terrible precision

- We often think that computers are more accurate than humans (or even error-free), but this could not be further away from the truth
- **Machine epsilon (`macheps`):** relative step size between two representable numbers, or $2^{-52} \approx 2.2 \cdot 10^{-16}$
  - Max. relative error: `macheps/2`
- Rounding errors (e. g. if numbers have different orders of magnitude), catastrophic cancellation, ill conditioning (high sensitivity to small input errors) make it **even worse**
- Do not forget about programmers' mistakes and bugs

Real case #1: numerical derivative failure

- Suppose that an economist is modelling Y that contains GDP as a linear term: $Y \coloneqq f(\text{GDP}, \ldots) + \varepsilon = GDP + f(\ldots)$
  - $\partial Y / \partial GDP = 1$, but they use numerical derivatives
- Lux GDP is 80 bn € $\Rightarrow$ the gap between two representable numbers is $8 \cdot 10^{10} / 2^{52} \approx 1.7 \cdot 10^{-5}$
- Default step size in derivative computation: 10^{-6}

$$\nabla_{\text{GDP}} Y \Big|_{\text{GDP}_{\text{Lux}}} \approx \frac{[8 \cdot 10^{10} + 10^{-6}] - 8 \cdot 10^{10}}{10^{-6}}$$

- However, $[8 \cdot 10^{10} + 10^{-6}] \overset{64}{\not=} 8 \cdot 10^{10} \Rightarrow$ zero difference in the numerator (because $10^{-6} < 1.7 \cdot 10^{-5}$)!
  - The computer returns 0 instead of 1 $\Rightarrow$ numerical error

How computers compute functions

- CPUs can only add and multiply (and flip bits based on some instructions)
- However, we need such functions as $\exp x$, $\sin x$, $\Gamma(x)$, $\Phi(x)$ (normal CDF)
- Which trick can be used to compute any function with addition and multiplication? Polynomial approximation
  - Weierstrass approximation theorem in action
  - Taylor expansion, Chebyshev polynomials, Remez algorithm for given intervals
- Pick desired accuracy, split $\mathbb{R}$ into intervals, find polynomials that achieve this accuracy

Example: computing normal CDF

$\Phi(x) \coloneqq 1/2 + \frac{1}{\pi}\int_0^x e^{-t^2/2}\mathrm{d}t$ – no analytical expression exists!

Taylor expansion at $x = 0$ (pure arithmetic):

$$\Phi(x) = 1/2 + (x - x^3/3 + x^5/10)/\sqrt{\pi} + O(x^7)$$

A more accurate one for $x \in [0, \infty)$ with $|\varepsilon| < 3 \cdot 10^{-5}$:

$$\Phi(x) = 1 - e^{-x^2}/\sqrt{2\pi} \cdot (1.3x^2 + 6.8x + 34)/(x+3)^3$$

- Caveat: e^t itself requires approximation: $\sum_{i=0}^{n} t^i/i!$
- A better solution: $p = t\log_2 e \Rightarrow e^t = \underbrace{2^{\lfloor p \rfloor}}_{\text{integer}} \cdot \underbrace{2^{p-\lfloor p \rfloor}}_{\substack{\text{polynomial} \\ \text{approx.}}}$

Reading the source code

The source code of programming languages is immensely useful in debugging when there is nothing else remaining.

Examples (check these links):

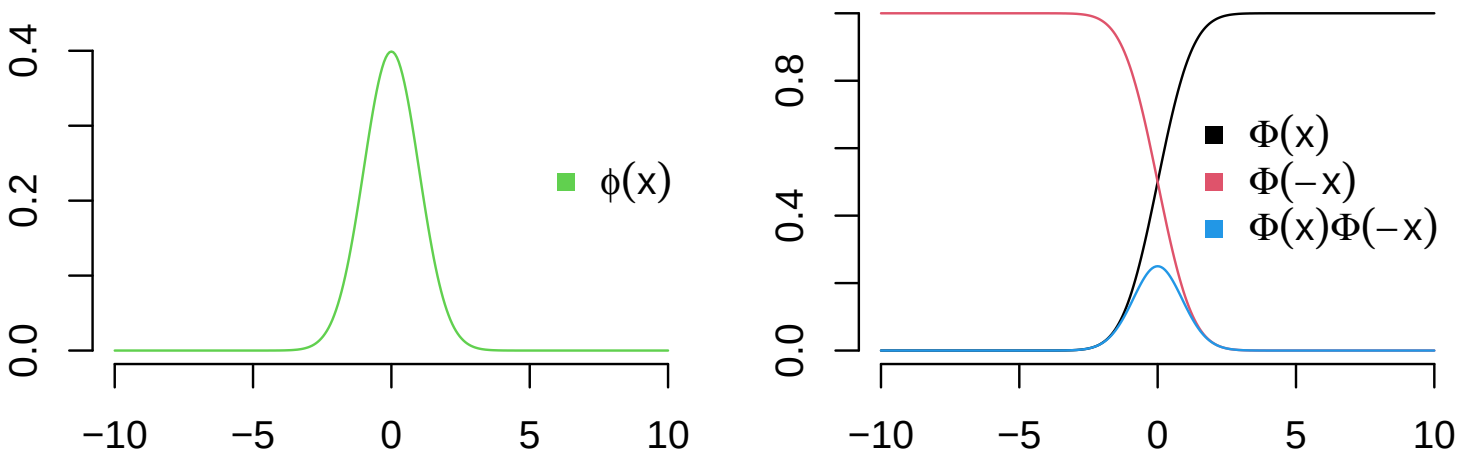
- Source of Gaussian CDF
- Source of gamma function $\Gamma(x)$
- Source of inverse Gaussian CDF $\Phi^{-1}(x)$
- Sine wave

Real case #2: dealing with ratios

A DEM doctoral student needed to compute

$$W(x) \coloneqq \frac{\phi(x)}{\Phi(x)\Phi(-x)},$$

where $\phi(x) \coloneqq (2\pi)^{-1/2} \exp(-x^2/2)$ is the Gaussian density, $\Phi(x) \coloneqq \int_{-\infty}^{x} \phi(t)\,\mathrm{d}t$ is its cumulative distribution function.

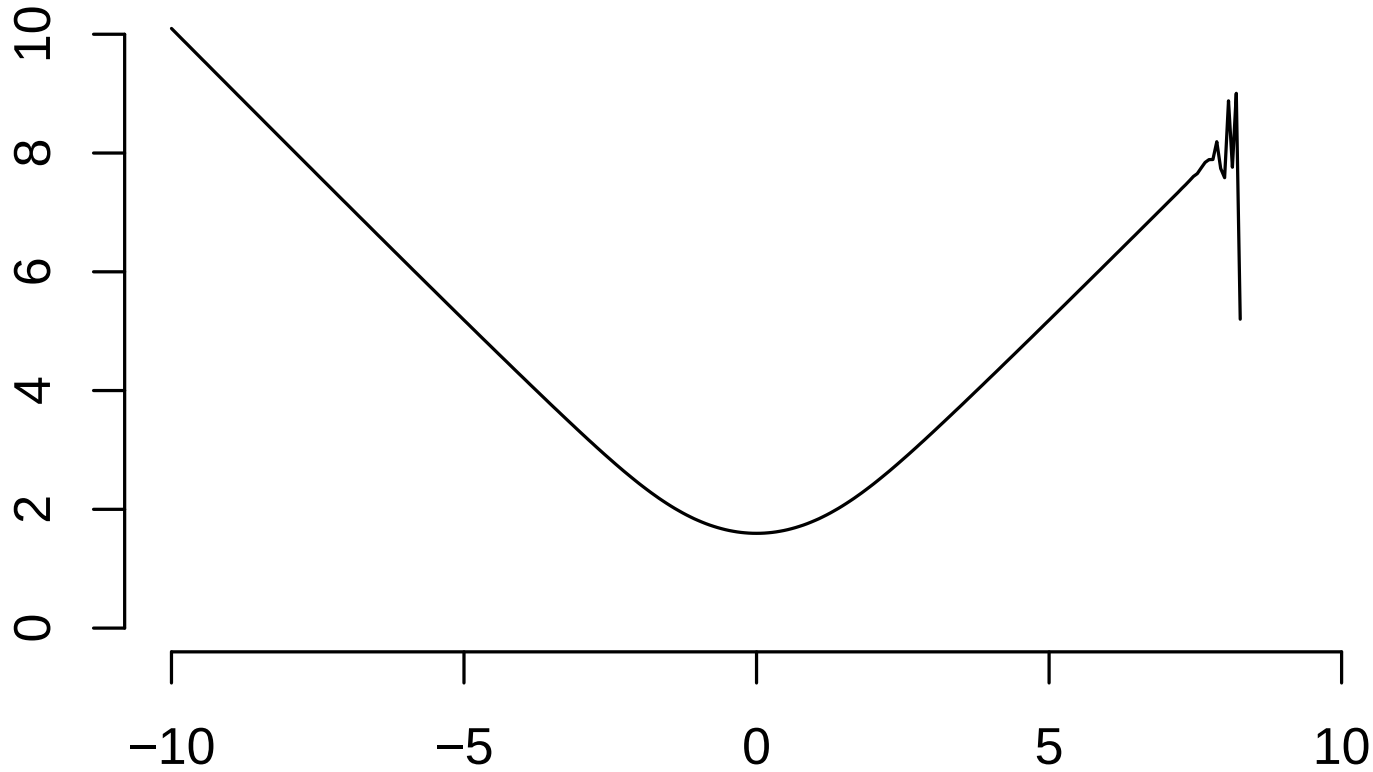


Goal: green function divided by cerulean function.

Real case #2: simple approach failure

Straightforward solution:

```
dnorm(x) / pnorm(x) / pnorm(-x)
# Same: dnorm(x) / (pnorm(x) * pnorm(-x))
```



Real case #2: root cause

The exponential function grows / decays so quickly, the numbers quickly approach the limits of machine precision.

Note that $\Phi(-x) = 1 - \Phi(x)$, but not on a computer.

At $x = 8.1$, the computer saves the following in the memory when computing the denominator:

- $\Phi(8.1) \overset{64}{\leadsto} 1 - 2.22 \cdot 10^{-16}$
- $\Phi(-8.1) \overset{64}{\leadsto} 2.75 \cdot 10^{-16}$ (but should be same as $1 - \Phi(x)$!)

The relative error is $\frac{2.75-2.22}{2.22} \approx 24\%$!

For larger $|x|$, the computer gives up: $\Phi(8.3) \overset{64}{\leadsto} 1$ **exactly**, i. e. $\Phi(8.3) - 1 = 0.000 \ldots 0$ in R (should be $\approx 5 \cdot 10^{-17} < \varepsilon/2$).

Real case #2: solution idea

Some developers (including R devs) added an alternative way of computing product involving rapidly growing / decaying functions without catastrophic accuracy loss.

Recall the relationship between '×', '+', and logarithms:

- $\log(a \cdot b/c) = \log a + \log b - \log c$
- $a \cdot b/c = \exp(\log a + \log b - \log c)$

Numerically unstable / near-zero functions usually have the option to return the logarithm instead:

$$\phi(x) = (2\pi)^{-1/2} e^{-x^2/2} \Rightarrow \log \phi(x) = -0.5 \log 2\pi - \frac{x^2}{2}$$

Only addition and multiplication ⇒ very stable!

Real case #2: solution implementation

Some R functions (densities, probabilities) have the option to return the **log** of the result:

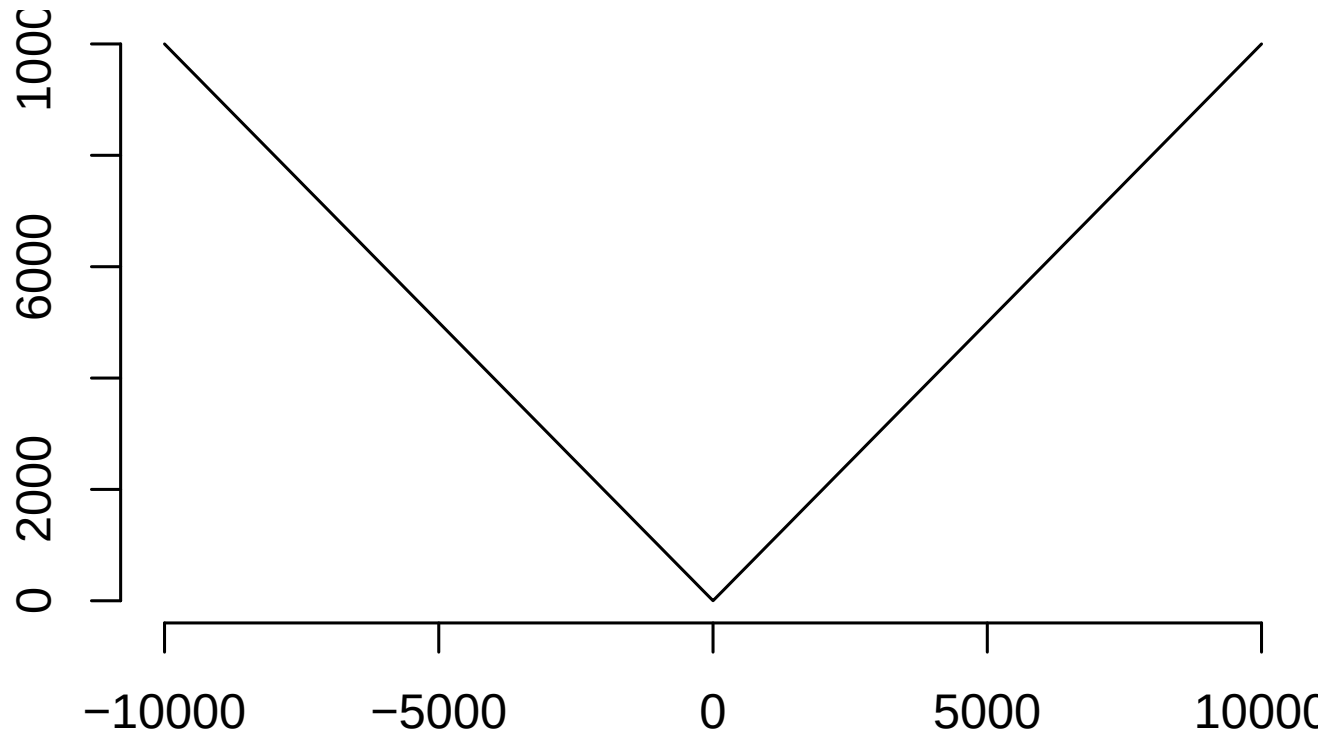
$$\phi(x)/\Phi(x)/\Phi(-x) = \exp(\log\phi(x) - \log\Phi(x) - \log\Phi(-x))$$

```
exp(dnorm(x, log = TRUE)
      - pnorm(x, log = TRUE)
      - pnorm(-x, log = TRUE))
```

Recall that $\log\phi(x) \propto x^2$, $\log\Phi(-|x|) \propto x^2$. As long as x^2 itself is accurate, its exponent is also reasonably accurate.

Real case #2: solution visualisation

```
s <- function(x) exp(dnorm(x, log = TRUE)
                     - pnorm( x, log = TRUE)
                     - pnorm(-x, log = TRUE))
```



Numerical accuracy summary

- Computers never store full real numbers – only their closest analogue on a variable-width grid
- This grid denser near zero, sparser for large numbers
- At any order of magnitude, requesting relative accuracy higher than $1.1 \cdot 10^{-16}$ is meaningless (unless one is using a custom arbitrary-precision solution)
- `(7/3 - 4/3) - 1 = 2.2e-16 = macheps`

Numerical and analytical equality

Consider 5 numbers computed in any 64-bit software:

```
a <- c(0.3, 0.4-0.1, 0.5-0.2, 0.6-0.3, 0.7-0.4)
length(unique(a))
```

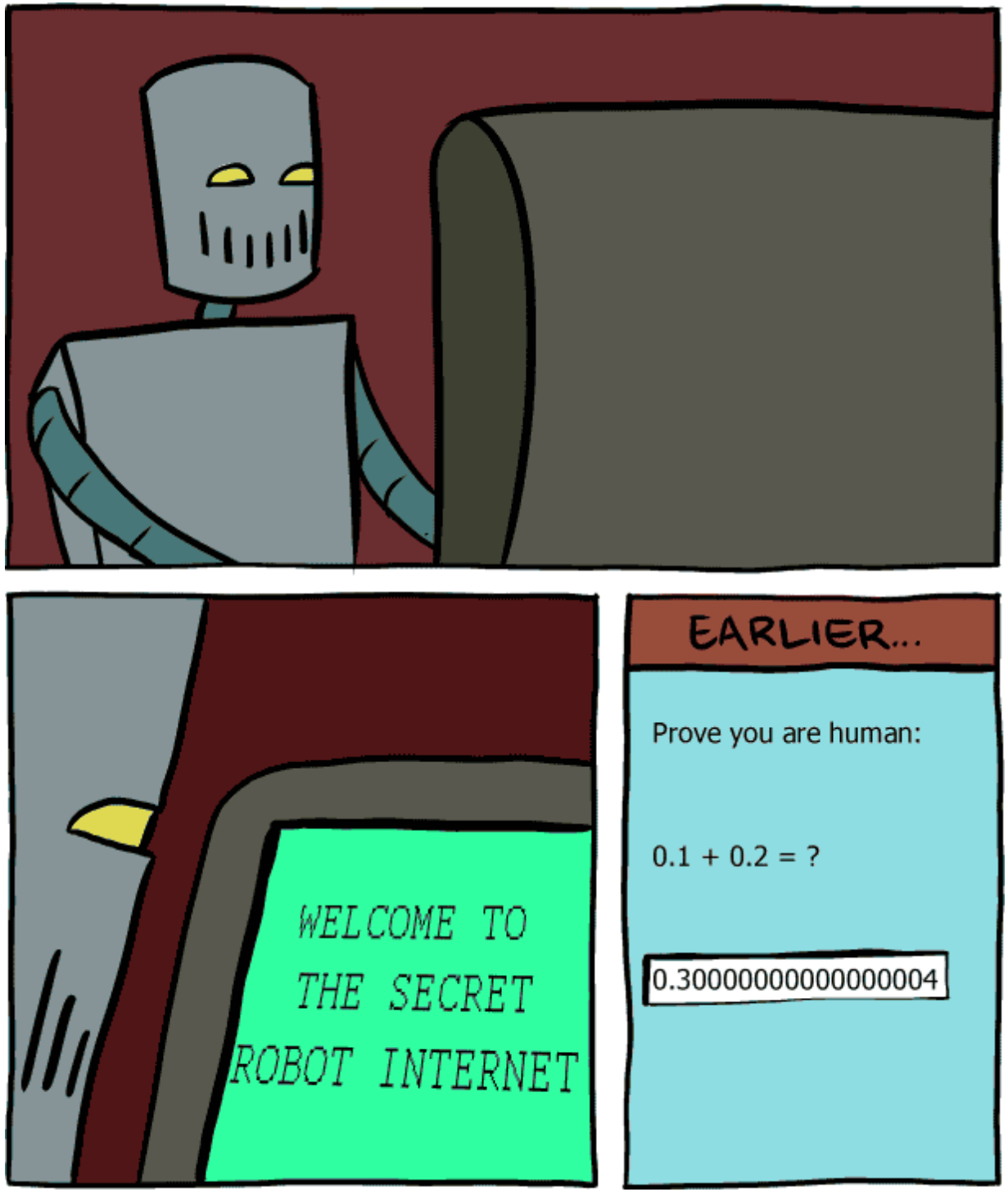
How many unique numbers are there?

Surprise: not 1. Not 5. But 3!

```
0.29..99, 0.30..04, 0.29..99, 0.29..99, 0.29..93
```

`..` denotes 13 repeated digits.

Try it in the Firefox console (F12).

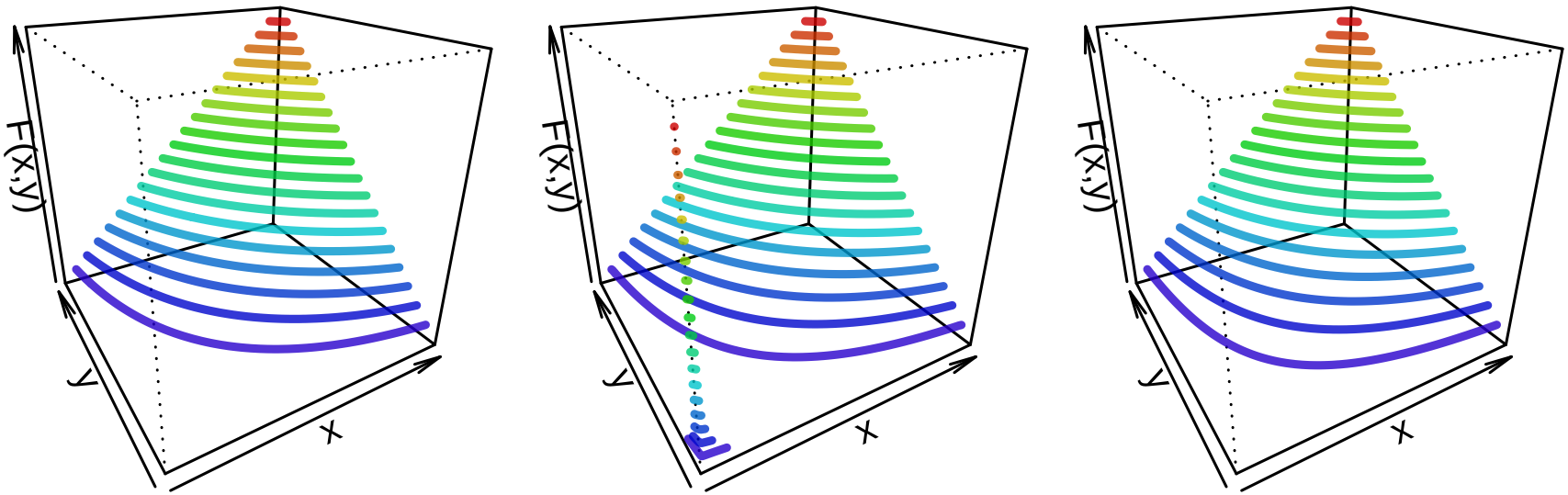

WELCOME TO
THE SECRET
ROBOT INTERNET
EARLIER...
Prove you are human:
0.1 + 0.2 = ?
0.30000000000000004

Credit: SMBC.

Real case #3: Estimation with copulæ

Visualising a function of 2 variables (Clayton copula):

$$F(x,y) \coloneqq \max\{(x^{-k} + y^{-k} - 1)^{-1/k}, 0\}, \quad k \in [-1, \infty) \setminus 0$$



$k = -0.35$ $k = -0.25$ $k = -0.15$

The weird spike does not appear if $k = -0.2499$ or -0.2501, but reappears if $k = -0.5$ or -0.125. **What is going on?**

Real case #3: Explanation

```
f <- function(x, y, k)
       pmax((x^(-k) + y^(-k) - 1)^(-1/k), 0)
x <- seq(0, 1, 0.01)
z <- outer(x, x, f, k = -0.25)
```

Let $b \coloneqq x^{-k} + y^{-k} - 1$. Problem: if $b < 0$ and $-1/k$ is not an integer, then, $b^{-1/k}$ is not well-defined!

The power function in C/C++, `pow` from `math.h`, for $b < 0$, succeeds if the exponent is integer (and fails otherwise). Integer powers are computed via repeated multiplication.

R, written in C, complies:

```
(-2)^c(-3.99,   -4, -4.01)
#>       NaN  0.0625  NaN
```

Real case #3: Precautions

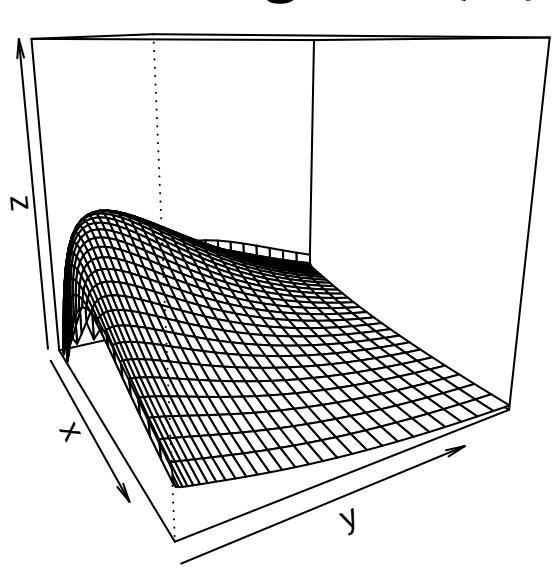
If there is a risk of your code creating an indeterminate or an ill-defined expression (0^0, $0/0$, $1^{1/0}$, $(1/0)/(1/0)$, $(1/0) - (1/0)$, $0 \times (1/0)$, b^p for $b < 0$ etc.),

1. Write a well-defined exception for the edge case;
2. Check if it **makes sense economically**.

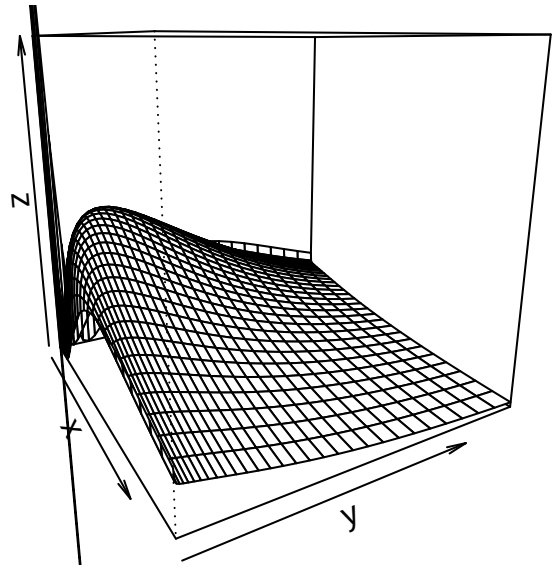
In this case, negative integer powers in the copula would make no sense for modelling financial returns because the densities obtained with the re-defined function look strange.

Real case #3: Practical conclusion

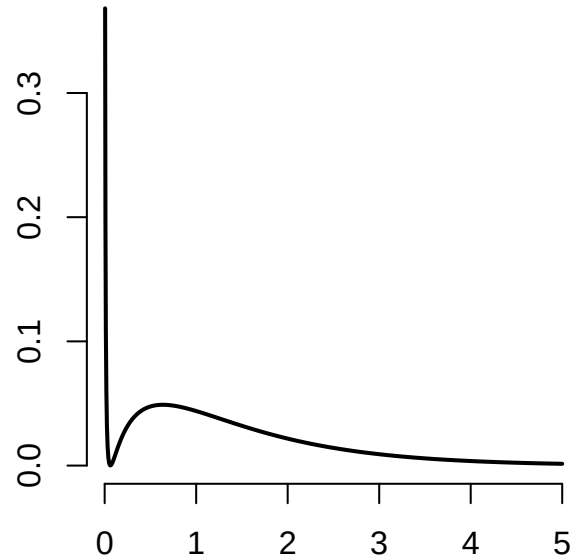
With integer $-1/k$, market return density may look erratic.



$k = -0.26$
3D view



$k = -0.25$
3D view



$k = -0.25$
side view

In this case, to avoid nonsensical subsequent analysis, re-define f by not allowing $-1/k > 0$ to be integer.

Programming concepts

Key to successful programming

Think like a computer!

It implies that one has to understand what is happening inside the black box.

> *'If you want to understand somthing, try explaining it to a computer.'*
>
> *Donald Erwin Knuth.*

Input and output (I/O)

Computers crunch inputs and return outputs.

- Mice: human movement as input, changes in the cursor position as output
  - Possible interpretation: pairs (x, y) of pixel values to draw the cursor at on the screen
- Programming languages: **streams** as input and output (sequences of data elements)
  - Programmes can operate independently, or require input from the user at the operating-system level
  - Programmes can read files or device signals as input and write files, blink lights, or play sounds as output

Piping

The output stream of one programme can be **piped** into another programme as input.

Example: (1) take the current time in nanoseconds, (2) trim the first 6 characters (keep the last 3), (3) divide by 2 and add 200, and (4) play through the speakers.

```
date +"%N" |              # Smth like 640713048
sed "s/^.\{6\}//" |       # 048
sed "s/$/\\/2+220/" |     # "048/2+220"
bc |                      # 244
awk '{print "ffplay -f lavfi -i \
 \"sine=frequency="$1":duration=1\" \
 -autoexit -nodisp"}' |   # Writes a command
eval "$(cat -)"           # Evaluates it
```

Sorting

Computers on their own cannot sort data (only read and write ones and zeros). They need a **rule** what to do with a string of numbers.

Example to sort n numbers (bubble sort):

1. Go from left to right $n - 1$ times
2. Check two adjacent numbers
3. If they are not in order, swap them

Strings are sorted alphabetically: [‘1’, ‘4’, ‘10’] will be sorted as [‘1’, ‘10’, ‘4’]!

- To ensure proper order, pad file names with zeros
- We have 10 sessions; this file is `session01.pdf`

Variables

Variables are **containers for storing values**. Upon variable creation, the computer reserves memory to store its contents (of any type).

Some languages (C, Java, Pascal) require declaring variable **type** (integer, string / character, array). R has **duck typing**:

> *If it walks like a duck and it quacks like a duck, then, it must be a duck.*

R can convert types dynamically.

Dynamic typing can be a curse

```
x <- c(6, -2, 2, -7)
print(x)
#> 6 -2  2 -7

sort(x)
#> -7 -2  2  6

sort(c(x, "John"))
#> "-2"   "-7"   "2"    "6"    "John"
```

- For applied economic analysis, in most cases, the desired data type is numeric matrix / array
- Text data requires special handling
- Image data (e. g. light intensity map) require even more special handling

Global and local variables

Global variables are accessible from any place of the programme by any function.

Local variables are created only at a function call, and are deleted afterwards (only the function return remains).

```
a <- 3          # Global a
f <- function(x) {
  b <- x^2      # Local b
  c <- b + a # By the way, this is bad practice
  return(c) # What if a is changed? Unreliable!
}
```

EViews problem: has no global variables – one needs to copy scalars from one page to another every time.

Conditions

By default, programmes are executed from top to bottom, in the order in which the lines go.

Conditions: control structures that check something and determine whether a block of code gets executed or skipped.

```
a <- 2
b <- 3
if (a < b) {
  print("Number b is greater than a.")
} else {
  print("Number a is greater or less than b.")
}
```

Loops

Iteration: repeating a block of commands many times, possibly with some changes.

'For' loop: iteration over *something*, i. e. running the same command using each element of *something* once.

'While' loop: iteration as many times as needed until something happens.

Loop termination

A ‘for’ loop runs predictably until it processes all values.

A ‘while’ loop can be potentially infinite – to avoid this, a **stopping criterion** is needed.

Common stopping rules:

- A change of something between two iterations is less than a small number (e. g. 10^{-8})
- Maximum number of iterations (e. g. 1000) reached
- Economics: The first derivative of something (usually he objective function) is close to zero (e. g. $|\nabla f| < 10^{-8}$ for all coordinates of f)

Array

Array: a data structure of fixed size that store elements of the same type.

Arrays can have arbitrary dimensions and are shaped like these objects:

- Vector
- Rectangle
- Rectangular cuboid (parallelepiped)
- Hypercube

Examples: string of text = 1D, spreadsheet = 2D, computer screen = 2D.

Array wrapping

Is a printed book a 1D or a 3D array? Is Proust's 'In Search of Lost Time' a 4D or a 5D array?

Arrangement into lines, columns, and pages: 3D.

- Errata refer to pages and lines to suggest corrections.

On the other hand, it is an ordered chain of characters that simply get **wrapped** around if they exceed the line length. Long stacks of lines are wrapped into pages (pagination).

Making books by wrapping lines is called **typesetting**.

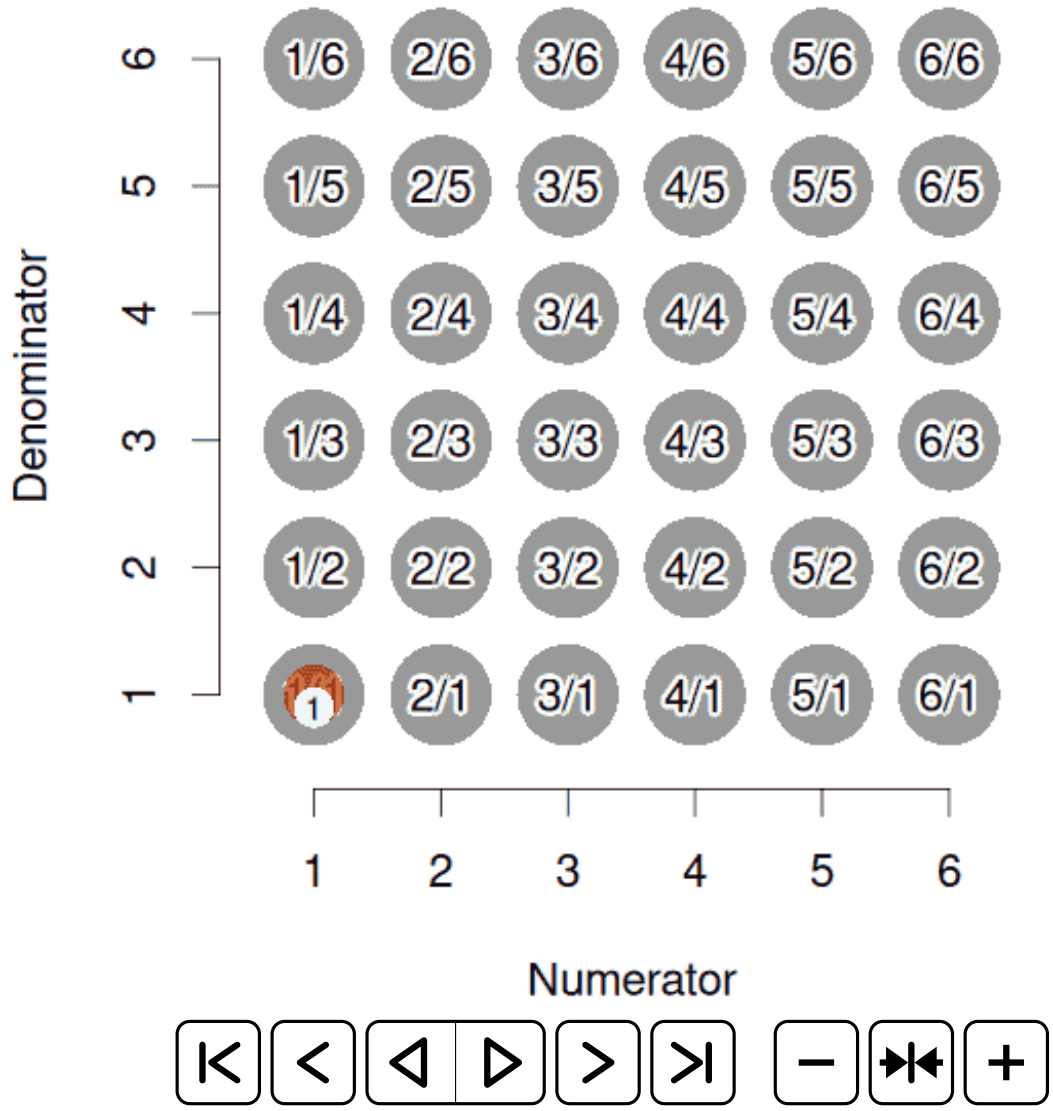
Is block G in Kirchberg a 1D or a 3D array?

Array indexing

An array with dimensions $d_1 \times d_2 \times \cdots \times d_n$ is representable via indices $1, 2, \ldots, (d_1 d_2 \cdots d_n)$.

Example: All rational numbers can be enumerated by a single integer.

Enumerating rationals with integers



Lists: when arrays are not enough

List: an abstract data type, a container that holds other objects (possibly heterogeneous).

R has out-of-the-box list support. Some languages may not support lists natively, but usually there is an extension for them (e. g. C++):

```
using namespace std;
using namespace boost;
typedef variant<string, int, bool> object;
struct vis : public static_visitor<> {
    void operator() (string s) const { /* */ }
    void operator() (int i) const { /* */ }
    void operator() (bool b) const { /* */ }
};
```

Object-oriented programming

OOP: An approach to programming where software design is centred around data and objects, as opposed to functions, logic, operations with bytes, and CPU instructions.

OOP involves applying specific methods to objects of specific classes.

An object can be a collection of other objects (values, arrays, strings, even functions), and methods are algorithms to perform specific operation on specific elements of the collection.

Example of OOP code (Python)

```
class Researcher:
  def __init__(self, name, food,
    field = "economics", coauthors = []):
    self.name = name
    self.field = field
    self.food = food
    self.coauthors = coauthors

  def print_greeting(self):
    print("Good morning, " + self.name + "!")

  def print_allstats(self):
    print(self.name + " is doing " + self.field +
      " and likes " + self.food + ".")
```

OOP code output (Python)

```
g117 = Researcher(name = "Andreas Irmen",
  food = "foie gras")
print(g117.name)
#> Andreas Irmen

g117.print_greeting()
#> Good morning, Andreas Irmen!

g117.print_allstats()
#> Andreas Irmen is doing economics and
#> likes foie gras.

g117.coauthors.append(["Anastasia Litina",
                       "Ka-Kit Iong"])
print(g117.coauthors)
#> [['Anastasia Litina', 'Ka-Kit Iong']]
```

Examining objects

In OOP languages, objects, being collections of certain values, can be examined in terms of structure. This is how one would 'unpack' everything that was put into an object:

```
print(g117.__dict__)
{'name': 'Andreas Irmen',
 'field': 'economics',
 'food': 'foie gras',
 'collaborators': [['Anastasia Litina',
                    'Ka-Kit Iong'
                   ]]
}
```

Non-OOP paradigms

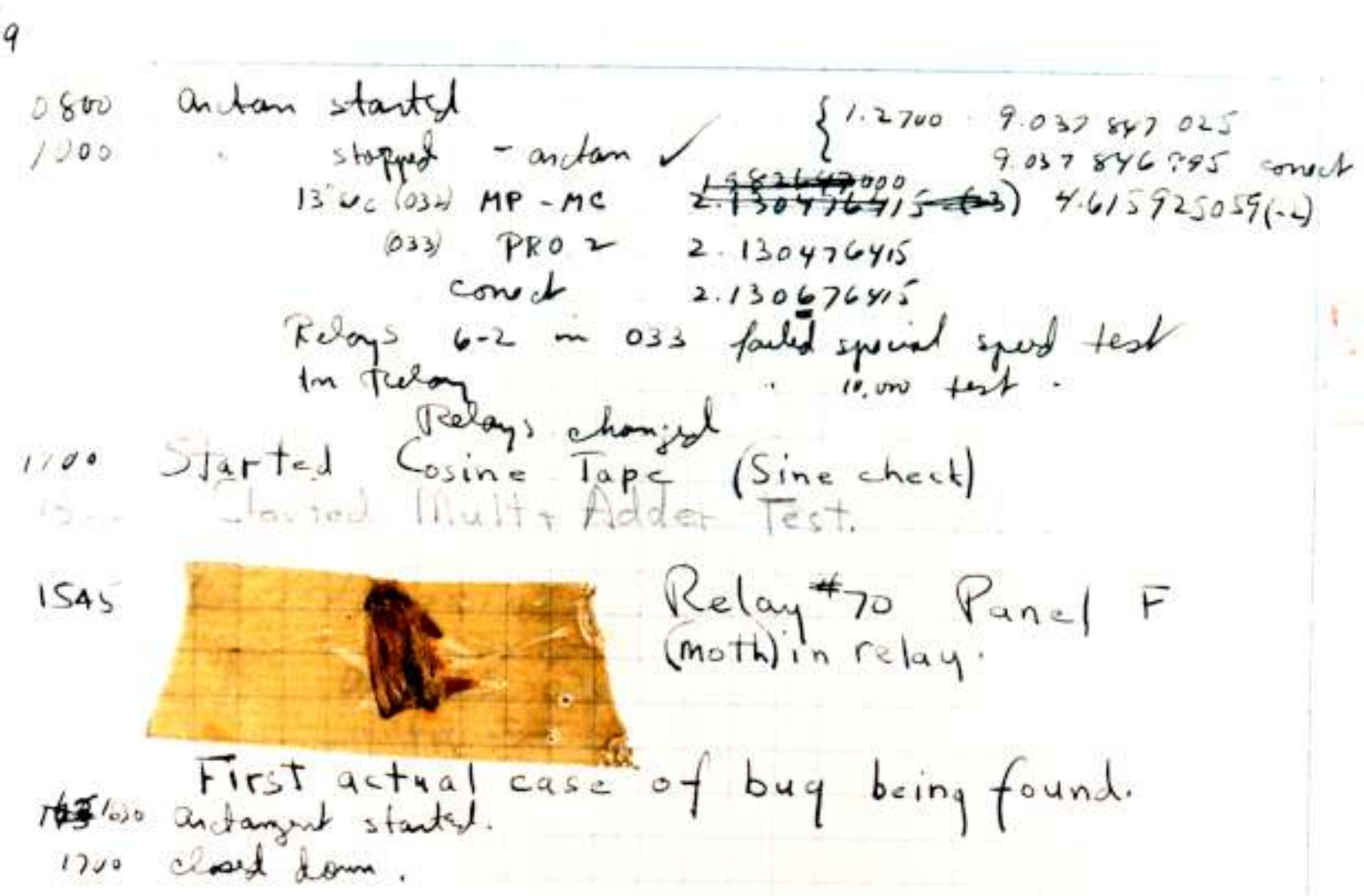
- Functional programming (almost everything is a fuction)
  - R is both FP and OOP
  - 'Everything that exists is an object, everything that happens is a function call.' – John Chambers on R.
- Imperative programming (programmes are lists of instructions)
  - Create variable *a* to store real numbers, set it equal to 3.4, initialise a variable for result, run this loop to update the result...
- Procedural programming (FORTRAN, C, Pascal)
  - Methods → procedures, object → record, class → module

Debugging

Troubleshooting your code, finding and eliminating **bugs** – mistakes that prevent your code from working as intended.

Tools for eliminating bugs:

- 'Print' (`print("Made it to here! 3")`)
- Debugger (step-by-step execution, memory debugger)
- Debugger in IDE
- Tracer (following the chain: which function called which)
- Linter (code analyser that checks the code without running it by checking syntax, variable use, and 'dangerous' features)



First actual case of bug being found: Mark II computer, Harvard University, engineering team of Grace Hopper (1947).

Compilation vs. interpretation

In R, one writes scripts that are **interpreted** on any machine running any OS (works out of the box):

```
e <- 2.718281828459045
print(e) # 2.718282
```

In Rust, one has to **compile** a programme (translate from human to machine code, i. e. processor instructions):

```
const E: f64 = 2.718281828459045;
fn main() { println!("{}", E); }
```

```
rustc -o mybin hello.rs
./mybin # 2.718281828459045
```

Compiled code example

```
hexdump hello
# 0000000 457f 464c 0102 0001 0000 0000 0000 0000
# 0000010 0003 003e 0001 0000 1040 0000 0000 0000
# <...>
# 0003a90 742e 7865 0074 662e 6e69 0069 722e 646f
# 0003aa0 7461 0061 652e 5f68 7266 6d61 5f65 6468
```

Compiled executables look different on different machines (PE/EXE on Windows, Mach-O on Mac, ELF on Unix/Linux).

Compiled executables may depend on external libraries containing functions. `R.exe` was compiled from thousands of C and Fortran files. Compiled binaries are usually **much faster**, but interpreted scripts are **more portable**.

Choose the right software

- Do not write dead code
- Do not choose the tools where the difficult parts have nothing to do with your research

We are in the process of digging ourselves into an anachronism by preserving practices that have no rational basis beyond their historical roots in an earlier period of technological and theoretical development.

Seymoure Papert, 1980.

Somewhere at SPSASsyViews, Inc.

Credit: Pieter Bruegel the Elder (1558).

Learn versatile tools

- Do no learn obscure or popular-but-limited packages
  - Cryptic software users may view themselves as priests with magical powers: ‘We are special, we can make it work’
  - In reality, mastering something pointlessly hard rather than making it easy perpetuates the hard nonsense
- R is not the only tool that is useful in your work – multiple open-source tools chained together are the best way to go: modular (not monolithic), always accessible, flexible
  - Example: pull PDF scans from server → recognise and extract text → write stemmed keywords by group into data set → run regression → generate colourful table
  - R may help you with the parts in green

R as a programming language

R architecture

- Base R: console interface (bare-bones command line)
- One can either type the commands into the console or save the commands as a script to execute as a whole
  - Similar to interactive vs. batch jobs on Uni.LU HPC in SLURM
- Interpreted programming language, scripts should run the same on Windows, Mac, Linux (unless there are dependencies on external libraries; rare)

Standard workflow

1. Open RStudio, the great editor and IDE
2. Write some code – a script that
   - Loads external data sources (Excel, CSV, text), pull the data from the web, clean and merge the data sets
   - Saves the transformed data for faster loading
   - Creates variables and functions (user objects), use them with the data to achieve the desired goal
   - Exports estimates, tables, images, useful outputs
3. Save the script and load it next time
   - Reproducible research: share the data and the script

Interacting with the outer world

- R can call other applications or interact with processes
  - Integrates with JDemetra+, EViews, or *any* piece of software that can be executed on that system
- Interaction with files via **connections**
  - Create or open files for writing
  - Pipe output to other processes
  - Read data via URLs from the Internet
- Uses **virtual devices** (similar to the physical monitor screen) for plotting when asked to save an image
  - PDF / PNG files are connections

Interaction with other software

- Can export to other formats thanks to user packages
- Can be called from within Python via `subprocess`
- Can be called from within EViews via a connection
  - Or vice versa, EViews can be called from within R
- Integration with gretl, Dynare
- C++ support out of the box with RTools

Beauty of R: multiple solutions

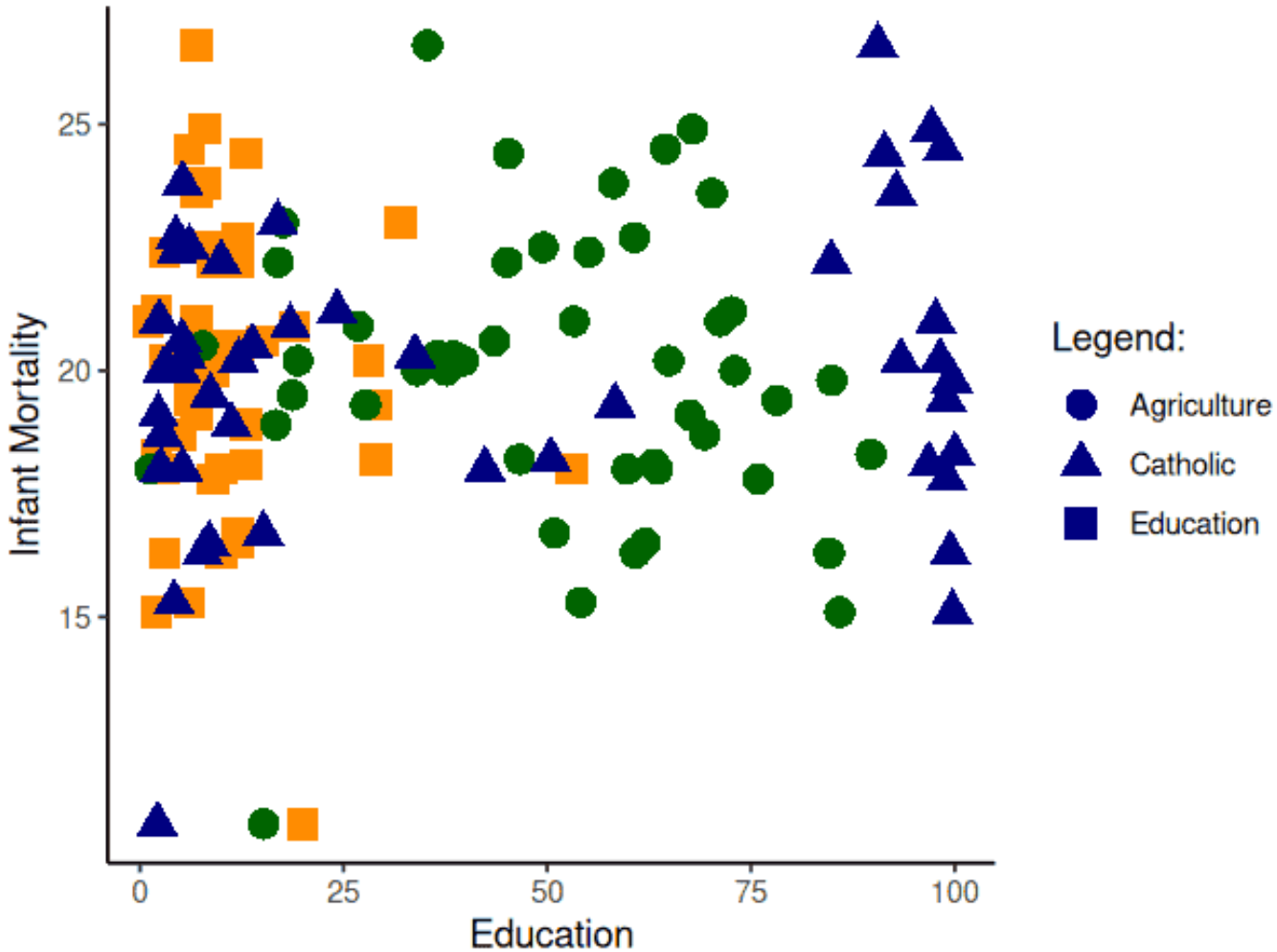
In R, there is no royal road to solving problems.

Various implementations might behave better or worse.

- Chain together certain operations into a function
- Find a package online with a function that does this (and test whether it is working as intended on a simple example)
  - Use multiple packages with various functions
- It is fine to be inefficient and sloppy with coding if it makes the workflow quicker (rapid proof-of-concept)
  - Your overall time (writing + execution) and code clarity (for others) are key!
  - However, if a certain part is redone / re-run multiple times, optimisation is desirable

Some R examples are sloppy (result)

This example was found online:



Credit: Worried-Bit5779 on Reddit.

Some R examples are sloppy (source)

The accompanying code came in one line (!), 612 bytes:

```
ggplot(swiss, aes(x = Education, y = Infant.Mortality)) +
  geom_point(aes(shape = "Education", fill = "Education"),
    size = 4, color = "darkorange") +
  geom_point(aes(x = Agriculture, shape = "Agriculture",
    fill = "Agriculture"), size = 4, color = "darkgreen") +
  geom_point(aes(x = Catholic, shape = "Catholic",
    fill = "Catholic"), size = 4, color = "navy") +
  scale_color_manual(values = c("navy", "darkorange", "darkgreen")) +
  scale_fill_manual(values = c("darkorange", "darkgreen", "navy")) +
  labs(x = "Education", y = "Infant Mortality", shape = "Legend:",
    color = "Legend:", fill = "Legend:") +
  theme_classic()
```

- R does not have to be ugly spaghetti code
- If it looks redundant, there should be a better solution
- Move away from the '1 result ← 1 command' paradigm

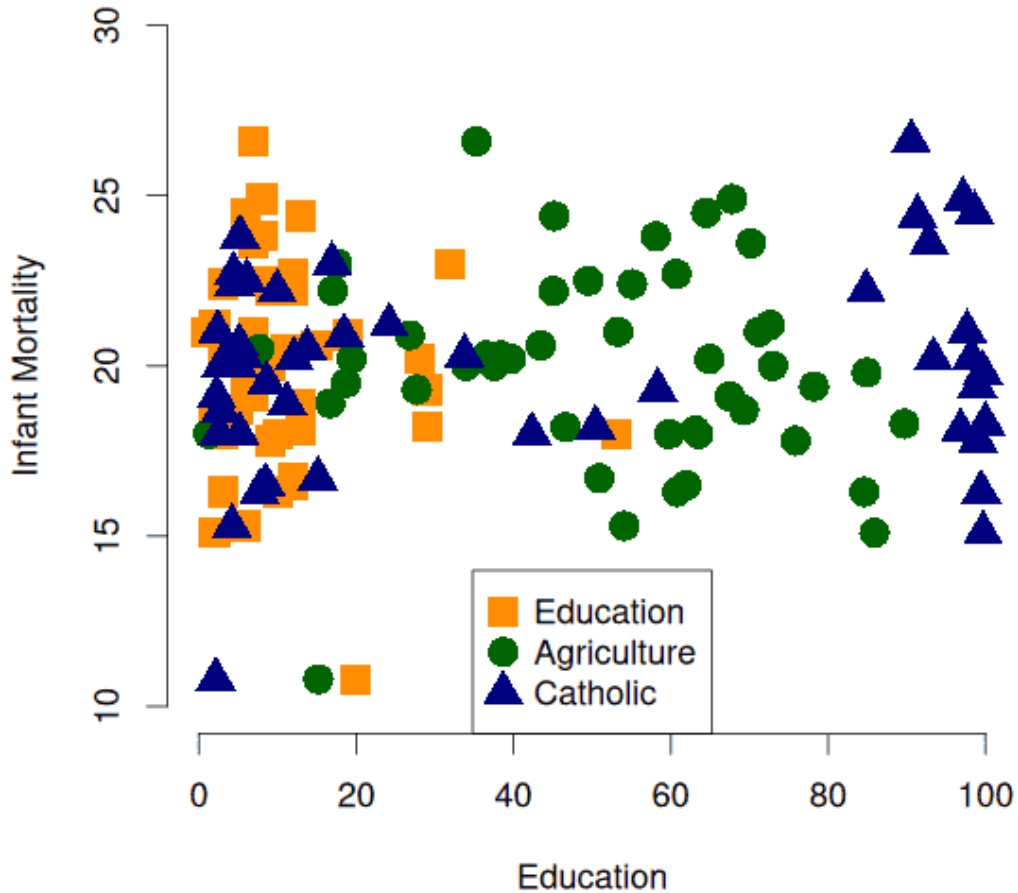
Power of base R: no packages required

612 → 421 characters thanks to removed redundancies and functions not doing extra unnecessary things that require clean-up.

```
y <- swiss$Infant.Mortality
xs <- c("Education", "Agriculture", "Catholic")
cls <- c("darkorange", "darkgreen", "navy")
plot(swiss[, xs[1]], y,
     pch = 15, col = cls[1], cex = 2, bty = "n",
     xlab = xs[1], ylab = "Infant Mortality",
     xlim = c(0, 100), ylim = c(10, 30))
points(swiss[, xs[2]], y, pch=16, col=cls[2], cex=2)
points(swiss[, xs[3]], y, pch=17, col=cls[3], cex=2)
legend("bottom", xs, col=cls, pch=15:17, pt.cex=2)
```

Plot produced by the cleaner code

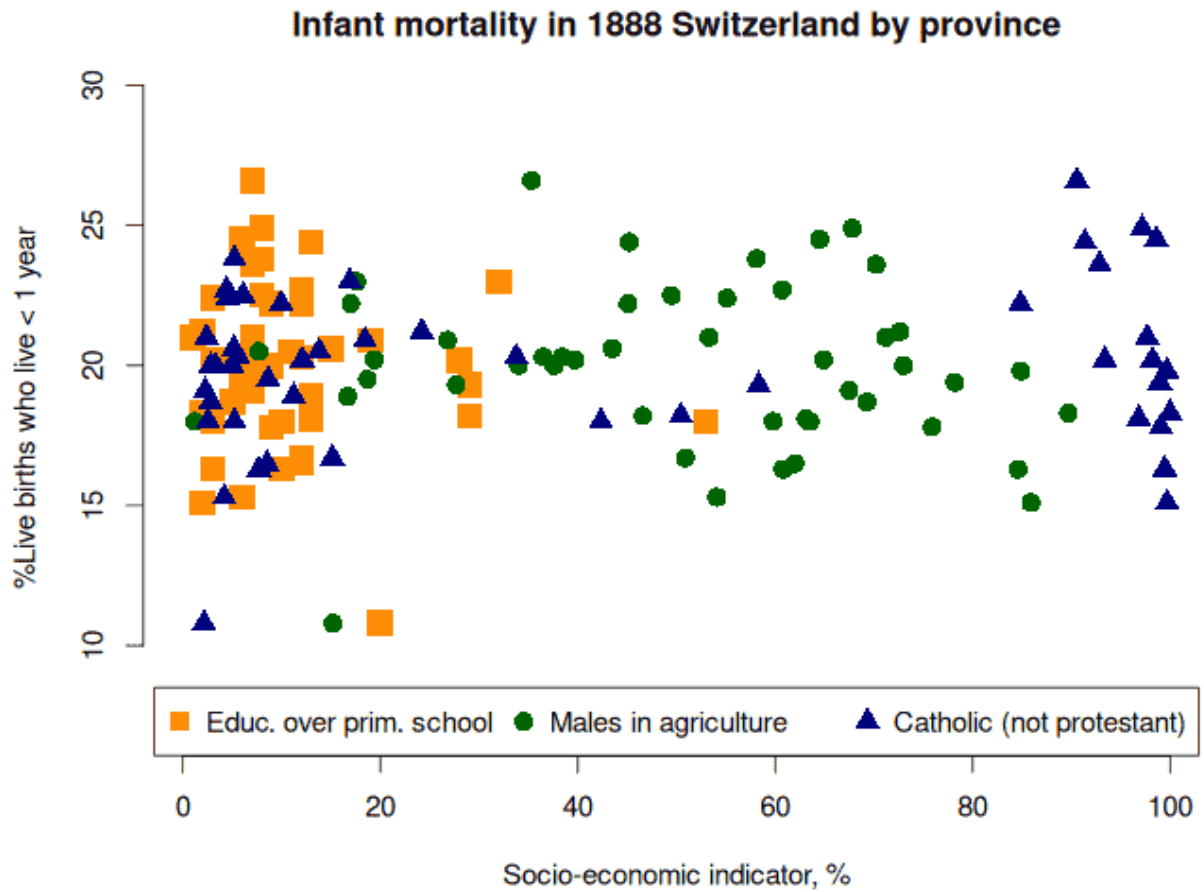
421 characters of base R achieve the same result:



Note that the legend has no colouring error.

One step further: adding readability

612 characters of base R can do more:



Can you tell us a story from this plot?

Do not overgolf

Code golf: recreational programming to fit a programme into the fewest bytes (real golf: fewest strokes win).

The same 421-byte code can be golfed into this 237-byte (!) unreadable abomination (producing an identical plot):

```
s=swiss;y=s[,6];v=s[,c(4,2,5)];l=c("#ff8c00","green4","na
plot(0,0,bty="n",xla="Education",yla="Infant Mortality",x
for(i in 1:3)points(v[,i],y,p=14+i,c=l[i],cex=2)
legend("bottom",names(v),co=l,pc=15:17,pt.c=2)
```

Further reading

- SPSS is dying. It's time to change
- Python in 2021: The Good, The Bad, and the Ugly
- Binary to decimal converer
- A Deep Dive Into How R Fits a Linear Model

Thank you for your attention!