Clarity-Driven Development of Scientific Software

Dr. Franziska Horn

2025-08-24

Table of contents

Preface

This book is meant to **empower researchers to code with confidence and clarity**.

If you studied something other than computer science—especially in the natural sciences like physics, chemistry, or biology—it's likely you were never taught how to properly develop software. Yet, you're often still expected to write code as part of your daily work. Maybe you've taken a programming course like *Python for Chemists* and can put together functional scripts through trial and error (with a little help from an AI assistant). But chances are, no one ever showed you how to write well-structured, maintainable, and reusable code that could make your life—and collaborating with your colleagues—so much easier.

This book is for you if you want to:

- Write functional software more quickly
- Use a structured approach to design better programs
- Reuse your code in future projects
- Feel confident about what your scripts are doing
- Prepare your research code for production
- Share your work with pride.

Whether you're just beginning your scientific journey—perhaps working on your first major project like a master's thesis or your first paper—or you're contemplating a move from academia to industry, the practical advice in this book can guide you along the way. We will approach software design from first principles and tackle research questions with a product mindset.

While the book contains some example code in Python to illustrate the concepts, the general ideas are independent of any programming language.[1]

Software development is a craft that's best learned with the guidance of a senior colleague—someone who can show you the right tools and provide feedback through code reviews. Unfortunately, mentors with industry experience are rare in academia. While a book can't replace an apprenticeship, I hope this one gives you a head start. It's the book I wish I could have read at university and the one I always wanted to recommend to the students and junior developers I've mentored.

Give Me Feedback, Please!

I'm always looking to improve the contents of this book (or any other resources you can find on my website). Please **send me an email** to `hey@franziskahorn.de`, if you have any suggestions for how this book could be improved!
Specifically, I'd really appreciate some **feedback** on

[1] Just in case, this tutorial provides a concise overview of the Python concepts used in this book.

- Where you got confused or lost or have an unanswered question
- Where you disagree, or have different experiences
- Where you started to get bored and felt like skipping ahead or giving up
- Anything you found especially interesting or helpful

Thank you so much & enjoy!

Acknowledgments

I would like to thank Marcel Lengert, Sarah Nomura, Ana Moga, and Robin Horn for their thoughtful feedback.

The texts in this book were partly edited and refined with the help of ChatGPT, however, all original content is my own.

How to Cite

```
@book{horn2025cddsci,
  author = {Horn, Franziska},
  title = {Clarity-Driven Development of Scientific Software},
  year = {2025},
  url = {https://franziskahorn.de/rsebook/},
}
```

Part I.

Gaining Clarity

Before you start writing code, it is important to gain clarity on your concept and approach (Figure 1). Specifically, in this first part of the book, we'll examine:

- Why you develop software — what problem you're trying to solve and what outcome you want to achieve (Chapter 1).
- What output your software should create to deliver the most value to your users (Chapter 2).
- How your code can generate this output, and how to get from working code to good code (Chapter 3).

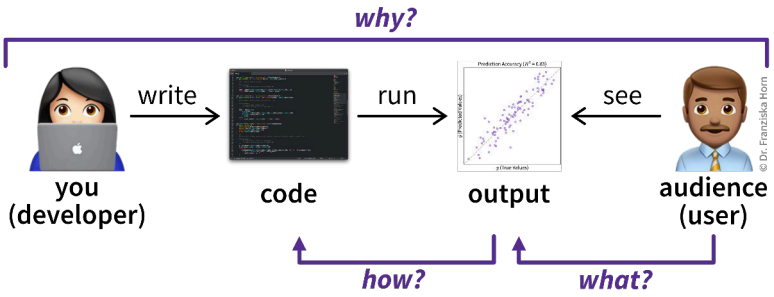


Figure 1.: A developer writes code, which is then executed to generate some output that is consumed by a user. Before you start writing this code, it is important to gain clarity on the why, what, and how of your solution.

1. Outcome: Why?

Before writing your first line of code, it's crucial to clearly understand what you're trying to achieve—specifically, the purpose of your research. This helps you focus on developing an innovative solution that creates real value by improving existing approaches or addressing unmet needs. Furthermore, this clarity will help you choose the most appropriate analysis methods to support your objectives and enable you to communicate your research effectively to ensure your audience understands the benefits of your work.

1.1. Why We Develop Software

While programming can be enjoyable in its own right, most people—and especially companies—aren't willing to invest significant time or resources unless there's a clear return. So why do we write code in the first place?

i Code vs. Software Product

Please note that there is a difference between ***code*** and a ***software product***:

- **Code** is simply **text written in a programming language** (like Python).
- A **software product** is code that actually **runs**—either locally (on your laptop or phone) or remotely (as a web service in the cloud)—and produces an **output** that **users interact with**.

Sometimes that interaction with the program is the end goal (e.g., when playing a video game). Other times, the software is just a means to an end—like a website you use to order clothes online or the script you run to generate your research results (Figure 1.1).

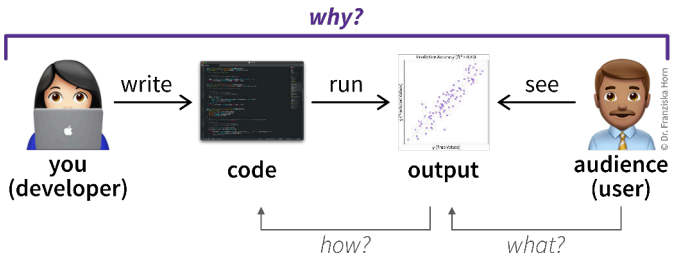


Figure 1.1.: To become valuable, code needs to be executed and produce some kind of output for a user. In this chapter we'll examine why we write this code in the first place, i.e., what outcome we're trying to achieve.

When it comes to serious software development, the motivation usually boils down to **profit** and/or **recognition**.

Companies are usually looking to make a profit, which can be accomplished in one of two ways:

1. **Increase revenue**: The company builds a software product that users are willing to pay for, like a web application offered as software-as-a-service (SaaS) with a monthly subscription, or a feature that results in customers spending more money, like better recommendations on an e-commerce platform.
2. **Reduce costs**: Alternatively, companies might build internal tools that automate tedious or repetitive tasks. By saving employee time, these tools reduce operational costs and indirectly increase profit.

As an individual—especially in research—your primary goal is probably to get some recognition in your field. For instance:

- You might write code to generate results that get your paper published in a respected journal and cited by others.
- Or you might create an open-source library that becomes popular (and receives a lot of stars on GitHub).

With a bit of luck, that recognition could also translate into profit, as a successful side project might lead to a lucrative job offer or form the basis for a new grant proposal.

1.2. Commercial Software Outcome: Satisfied Users

Whatever your motivation, success—whether financial or reputational—only comes if your software meets a **real need**. In other words, it must **create value for your users**. Let's first examine what this means for commercial software applications.

Before developing a product, we need to understand **our users and their priorities** [1]. These considerations are not only relevant for software, but apply to all kinds of products—physical and digital—like a woodworking tool or an e-commerce website.

Ask yourself:

1. **Who are your users?**
   Depending on the product, your target user group might be broad or highly specific. For example, a general consumer product could be used by anyone over 14, while enterprise solutions may cater to niche audiences, such as professionals in a particular field. Even if your users could theoretically be "everyone," picturing a more **specific user** can help refine your solution. Trying to please everyone often results in satisfying no one. Focusing on a distinct user group can also help differentiate your product in the market.

   User experience (UX) designers often create *user personas*—fictional but representative users based on real-world insights. These personas include details like age, profession, hobbies, and specific needs:

- *Woodworking tool:* "Mary, the multitasking mom"—a part-time teacher who enjoys DIY projects and wants to build a bird house with her daughter.
- *E-commerce website:* "Harry, the practical shopper"—a 55-year-old lawyer who wants to buy a birthday gift for his partner.

2. **What are their priorities?**
   What is important to your target user? Why are they looking for alternatives to existing solutions?

   - *Woodworking tool:* Mary mainly cares about the weight and noise level of the tool—it should be light enough for one-handed use and quiet enough to be used inside an apartment in the city.
   - *E-commerce website:* Harry wants to complete his task quickly, so he values a clean, easy-to-navigate design and the ability to find suitable products with minimal effort.

Learning about your users and their priorities can give you a clearer sense of where to focus your efforts. If current solutions fall short in the dimensions your users care most about, then you've **identified a meaningful gap**—a real problem that's worth solving.

The next step is to explore an **innovative idea**: a way to address this problem **more effectively than existing alternatives**, at least for this specific group of users. You may not yet know whether such a solution is technically feasible, but the gap itself justifies further exploration to better satisfy your users' needs.

1.3. Research Outcome: Knowledge Gain

In the world of consumer products, an innovative solution often addresses unmet needs or improves a frustrating and inefficient user experience. In research, we also aim to **advance the state of the art**. That might mean filling a gap in knowledge, or developing a new method, material, or process with improved properties. In your area of expertise, you're probably already aware of something that could be improved—where existing approaches fall short and where your idea might offer a better solution.

Research goals are often shaped by the analytical methods we use, so clarifying the type of question you're addressing can sharpen your focus (Section 1.3). While research doesn't have "users" in the commercial sense, our work is still judged by peers. To convince them of its value—e.g., to get a paper accepted—we must demonstrate that our approach outperforms existing ones on the criteria that matter. For this, we rely on evaluation metrics that help quantify our idea's advantages (Section 1.3).

Types of Research Questions

Most research questions can be categorized into four broad groups, each associated with a specific type of analytics approach (Figure 1.2).

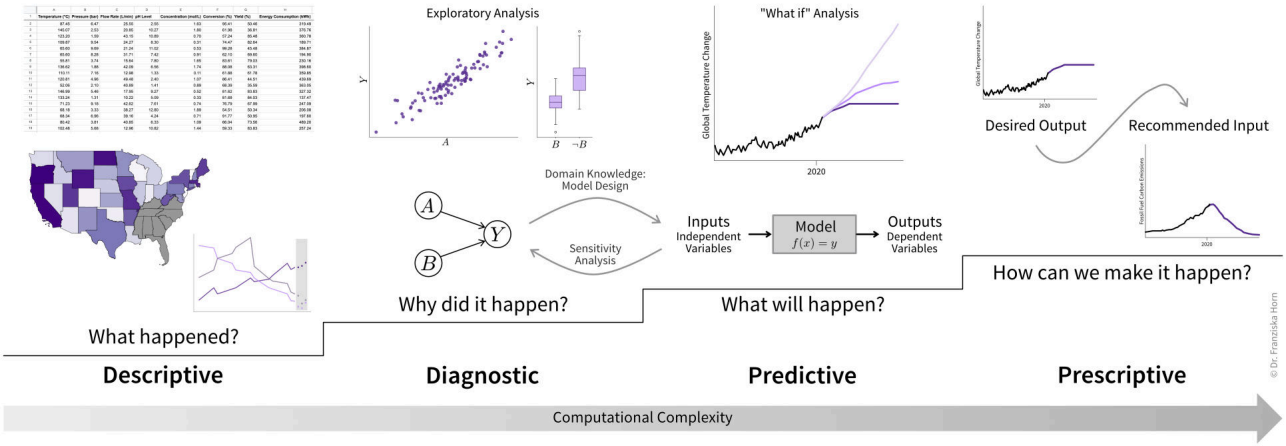


Figure 1.2.: Descriptive, diagnostic, predictive, and prescriptive analytics, with increasing computational complexity and need to write custom code.

Descriptive Analytics

This approach focuses on observing and describing phenomena to establish baseline measurements or track changes over time.
Examples include:

- Identifying animal and plant species in unexplored regions of the deep ocean.
- Measuring the physical properties of a newly discovered material.
- Surveying the political views of the next generation of teenagers.

Methodology:

- Collect a large amount of data (e.g., samples or observations).
- Calculate summary statistics like averages, ranges, or standard deviations.

Diagnostic Analytics

Here, the goal is to understand relationships between variables and uncover causal chains to explain *why* phenomena occur.
Examples include:

- Investigating how CO_2 emissions from burning fossil fuels drive global warming.
- Evaluating whether a new drug reduces symptoms and under what conditions it works best.
- Exploring how economic and social factors influence shifts toward right-wing political parties.

Methodology:

- Perform exploratory data analysis, such as looking for correlations between variables.
- Conduct statistical tests to support or refute hypotheses (e.g., comparing treatment and placebo groups).
- Design of experiments to control for external factors (e.g., randomized clinical trials).

- Build predictive models to simulate relationships. If the predictions from these models match new real-world observations, it suggests their assumptions correctly represent causal effects.

Predictive Analytics

This method involves building models to describe and predict relationships between independent variables (inputs) and dependent variables (outputs). These models often rely on insights from diagnostic analytics, such as which variables to include in the model and how they might interact (e.g., linear or nonlinear dependence). Despite its name, this approach is not just about predicting the future, but used to estimate unknown values in general (e.g., variables that are difficult or expensive to measure). It also includes any kind of simulation model to describe a process virtually (i.e., to conduct *in silico* experiments).
Examples include:

- Weather forecasting models.
- Digital twin of a wind turbine to simulate how much energy is generated under different conditions.
- Predicting protein folding based on amino acid sequences.

Methodology:
The key difference lies in how much domain knowledge informs the model:

- *White-box (mechanistic) models:* Based entirely on known principles, such as physical laws or experimental findings. These models are often manually designed, with parameters fitted to match observed data.
- *Black-box (data-driven) models:* Derived primarily from observational data. Researchers usually test different model types (e.g., neural networks or Gaussian processes) and choose the one with the highest prediction accuracy.
- *Gray-box (hybrid) models:* These combine mechanistic and data-driven approaches. For example, the output of a mechanistic model may serve as an input to a data-driven model, or the data-driven model may predict residuals (i.e., prediction errors) from the mechanistic model, where both outputs combined yield the final prediction.

> Resources to learn more about data-driven models
>
> If you want to learn more about how to create data-driven models and the machine learning (ML) algorithms behind them, these two free online books are highly recommended:
>
> – [11] Supervised Machine Learning for Science by Christoph Molnar & Timo Freiesleben; a fantastic introduction focused on applying black-box models in scientific research.
> – [13] A Practitioner's Guide to Machine Learning by me; a broader overview of ML methods for a variety of use cases.

Provided the developed model is sufficiently accurate, researchers can then analyze its behavior (e.g., through a sensitivity analysis, which examines how outputs change with varying inputs) to gain further insights about the modeled system itself (to feed back into diagnostic analytics).

Prescriptive Analytics

This approach focuses on decision-making and optimization, often using predictive models. Examples include:

- Screening thousands of drug candidates to find those most likely to bind with a target protein.
- Optimizing reactor conditions to maximize yield while minimizing energy consumption.

Methodology:

- *Decision support:* Use models for "what-if" analyses to predict outcomes of different scenarios. For example, models can estimate the effects of limiting global warming to 2°C versus exceeding that threshold, thereby informing policy decisions.
- *Decision automation:* Use models in optimization loops to systematically test input conditions, evaluate outcomes (e.g., resulting predicted material quality), and identify the best conditions automatically.

> **!** Model accuracy is crucial
>
> These recommendations are only as good as the underlying models. Models must accurately capture causal relationships and often need to extrapolate beyond the data used to build them (e.g., for disaster simulations). Data-driven models are typically better at interpolation (predicting within known data ranges), so results should ideally be validated through additional experiments, such as testing the recommended new materials in the lab.

Together, these four types of analytics form a powerful toolkit for tackling real-world challenges: descriptive analytics provides a foundation for understanding, diagnostic analytics uncovers the causes behind observed phenomena, predictive analytics models future scenarios based on this understanding, and prescriptive analytics turns these insights into actionable solutions. Each step builds on the previous one, creating a systematic approach to answering complex questions and making informed decisions.

Evaluation Metrics

To demonstrate the impact of your work and compare your solution against existing approaches, it's crucial to define what success looks like quantitatively. Consider these common evaluation metrics to measure the outcome of your research and generate compelling results:

- **Number of samples:** This refers to the amount of data you've collected, such as whether you surveyed 100 or 10,000 people. Larger sample sizes can provide more robust and reliable results. It is also important to make sure your sample is representative of the population as a whole, i.e., to avoid sampling bias, which can cause misleading results and incorrect conclusions.
- **Reliability of measurements:** This evaluates the consistency of your data. For example, how much variation occurs if you repeat the same measurement, e.g., run a simulation with different random seeds. This is important as others need to be able to reproduce your results.

- **Statistical significance:** The outcome of a statistical hypothesis test, such as a p-value that indicates whether the difference in symptom reduction between the treatment and placebo groups is significant.
- **Model accuracy:** For predictive models, this includes:
  - Standard metrics like R^2 to measure how closely the model's predictions align with observational data.
  - Cross-validation scores to assess performance on new data.
  - Uncertainty estimates to understand how confident the model is in its predictions.
- **Algorithm performance:** This includes metrics like memory usage and the time required to fit a model or make predictions, and how these values change as the dataset size increases. Efficient algorithms are crucial when scaling to large datasets or handling complex simulations.
- **Key Performance Indicators (KPIs):** Any other practical measures that matter in your field. For example:
  - For a chemical process: yield, purity, energy efficiency.
  - For a new material: strength, durability, cost.
  - For an optimization task: convergence time, solution quality.

Your evaluation typically involves multiple metrics. For example, in prescriptive analytics, you need to demonstrate both the accuracy of your model and that the recommendations generated with it led to a genuinely optimized process or product. Before starting your research, **review similar work in your field to understand which metrics are standard in your community**.

1.4. Draw Your Idea

Whether you're collaborating with colleagues, presenting at a conference, or writing a paper—it is essential to clearly communicate the problem you're solving and your proposed solution.

Visual representations are particularly powerful for conveying complex ideas. One effective approach is creating **"status quo vs. your contribution" visuals that contrast the current state of the field with your proposed improvements** (Figure 1.3).

The "status quo" depiction might show a lack of data, an incomplete understanding of a phenomenon, poor model performance, or an inefficient process or material. The "your contribution" diagram highlights how your research addresses these issues and improves on the current state, such as refining a predictive model or enhancing the properties of a new material.

At this point, the diagram showing "your contribution" might be based on a hypothesis or an educated guess about what your results will look like—and that's totally fine! The purpose of visualizing your solution is to guide your development process. Later, you can update the picture with actual results if you decide to include it in a journal publication, for example.

Of course, not all ideas are tied directly to analytics. Sometimes the main improvement is more qualitative, for example, focusing on design or functionality (Figure 1.4).

A "status quo" illustration is not always necessary, for example, if existing solutions are not directly comparable or so well known that they require no further explanation (Figure 1.5).

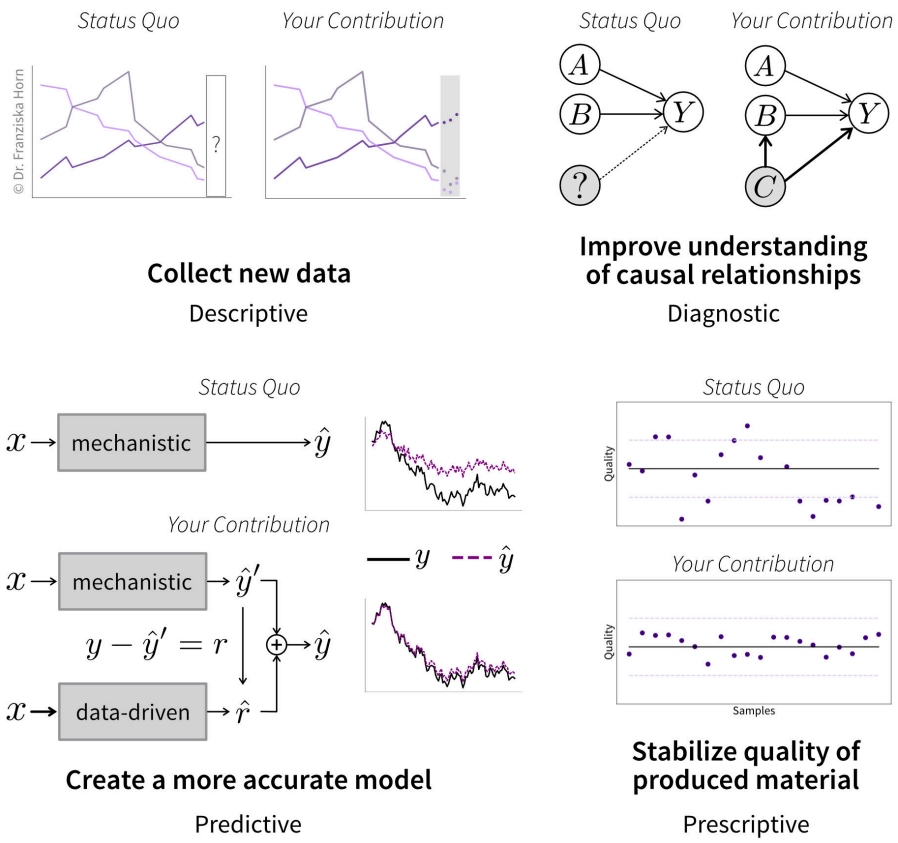


Figure 1.3.: Exemplary research goals and corresponding "status quo vs. your contribution" visuals for descriptive, diagnostic, predictive, and prescriptive analytics tasks.

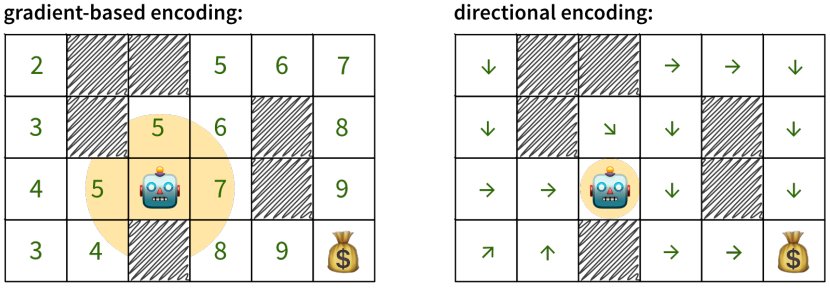


Figure 1.4.: This example illustrates a task where a robot must reach its target (represented by money) as efficiently as possible. **Original approach *(left)***: The robot relied on information encoded in the environment as expected rewards. To determine the shortest path to the target, the robot required a large sensor (shown as the yellow circle) capable of scanning multiple nearby fields to locate the highest reward. **New approach *(right)***: Instead of relying on reward values scattered across the environment, the optimal direction is encoded directly in the current field. This eliminates the need for large sensors, as the robot only needs to read the value of its current position, enabling it to operate with a much smaller sensor and thereby reducing hardware costs. ***Additional experiments*** still need to demonstrate that with the new approach, the robot reaches its target at least as quickly as with the original approach.

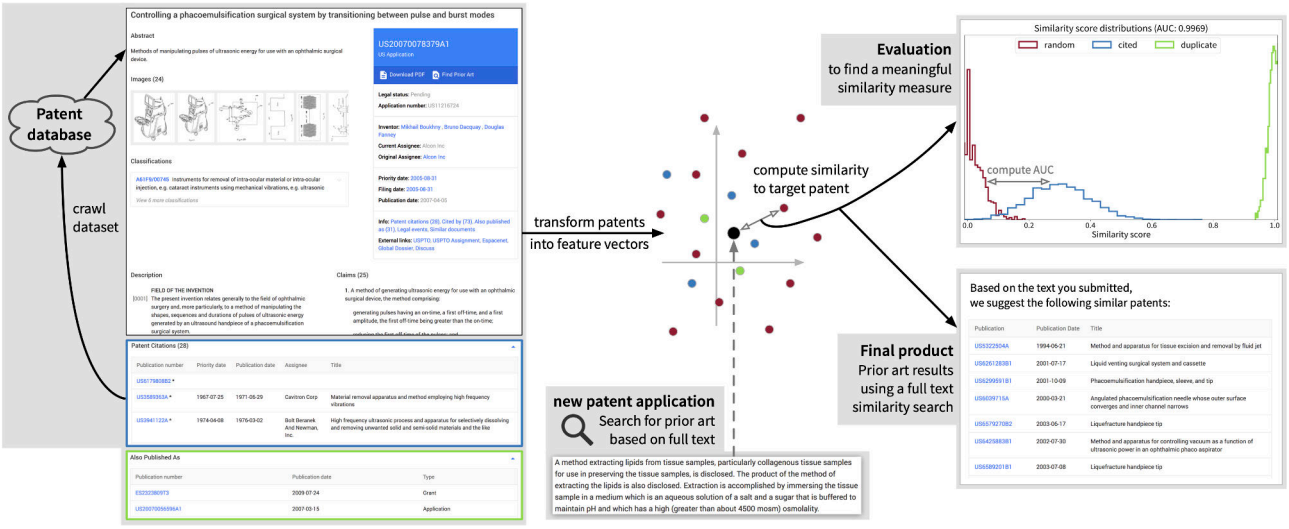


Figure 1.5.: Diagram of the approach used in the paper *Automating the search for a patent's prior art with a full text similarity search* [12] (where the alternative would be an ordinary manual keyword search).

Of course, the examples shown here are already refined for publication—your initial sketches will probably look a bit messier (Figure 1.6). Have a look at [4] for some tips on communicating science through visualizations and creating insightful graphics.

Give it a try—does the sketch help you explain your research to your family?

1.5. Draw Your Advantage

Ideally, you should already have an idea of how existing approaches perform on relevant evaluation metrics (e.g., based on findings from other publications) to establish **the baseline your solution should outperform**. You'll likely need to replicate at least some of these baseline results (e.g., by reimplementing existing models) to ensure your comparisons are not influenced by external factors. But understanding where the "competition" stands can also help you identify secondary metrics where your solution could excel. For example, even if there's little room to improve model accuracy, existing solutions might be too slow to handle large datasets efficiently (Figure 1.7).[1]

These results are central to your research (and publications), and much of your code will be devoted to generating them, along with the models and simulations behind them. Of course, at this stage, your solution's performance on these dimensions is still aspirational—this graph simply **illustrates the gap your approach aims to fill**. But clearly defining the key metrics needed to demonstrate your research's impact will help you focus your programming efforts effectively.

[1]For example, currently, a lot of research aims to replace traditional mechanistic models with data-driven machine learning models, as these enable significantly faster simulations. A notable example is the AlphaFold model, which predicts protein folding from amino acid sequences—a breakthrough so impactful it was recognized with a Nobel Prize [3]!

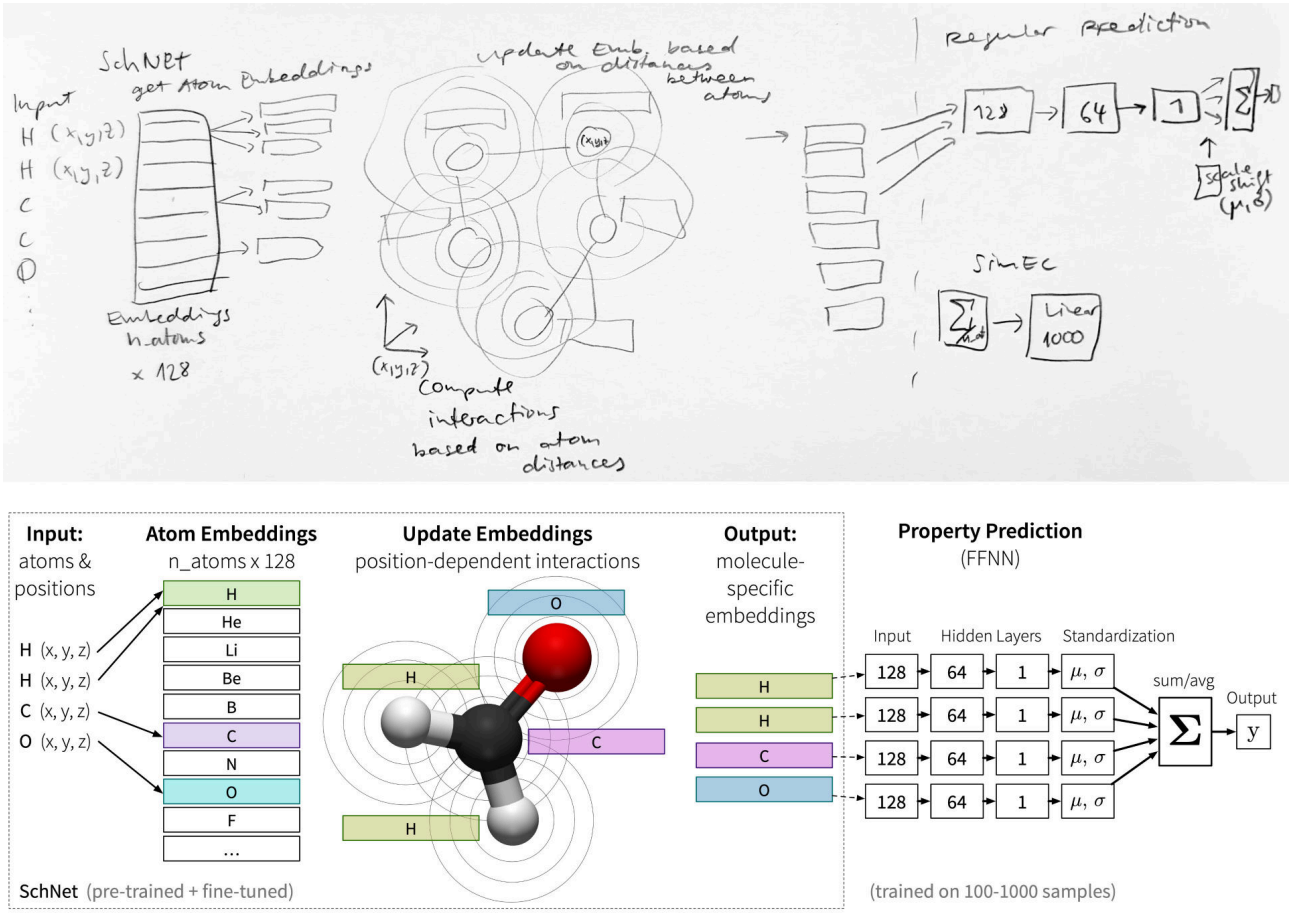


Figure 1.6.: One of several sketches and the resulting final figure that was included in my PhD thesis [14] (showing the SchNet neural network architecture).

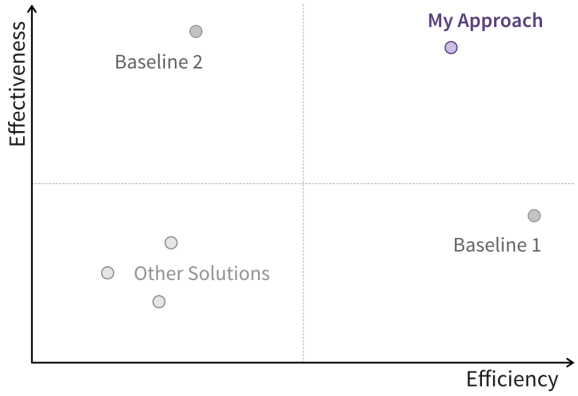


Figure 1.7.: **"Chart your competitive position"** [1]: The metrics we're interested in often represent trade-offs. For example, we want a high quality product, but it should also be cheap. Or a good model accuracy, but at the same time not use excessive compute resources. Your solution might not outperform existing baselines on all metrics, but its trade-off could still be preferable.

Before you continue

At this point, you should have a clear understanding of:

- The problem you're trying to solve.
- Existing solutions to this problem, i.e., the baseline you're competing against.
- Which metrics should be used to quantify your improvement on the current state.

2. Output: What?

In the previous chapter, we've gained clarity on the problem you're trying to solve and how to quantify the improvements your research generates. Now it's time to dive deeper into what these results might actually look like and the data on which they are built. More specifically, we want to understand what kind of output our code should create in order to be useful for our users (Figure 2.1), for example, what kind of plots would help your audience to understand the benefits of your approach.

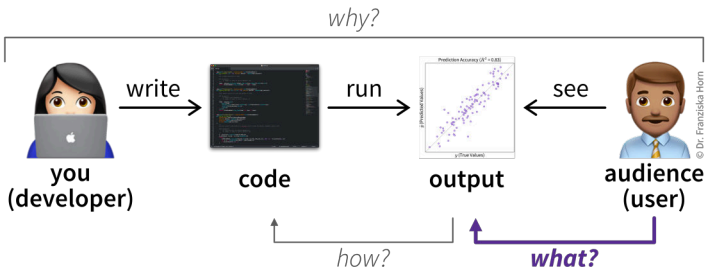


Figure 2.1.: The output that our code generates needs to be useful for our users.

2.1. Commercial Software Output: UX Design

Commercial software applications are often complex systems with interactive graphical user interfaces (GUIs) that must be carefully designed to meet user needs. To achieve this, **user experience (UX) designers typically create sketches, wireframes, or mockups** (Figure 2.2) to explore different design ideas and converge on a solution that is valuable, intuitive, and enjoyable to use.

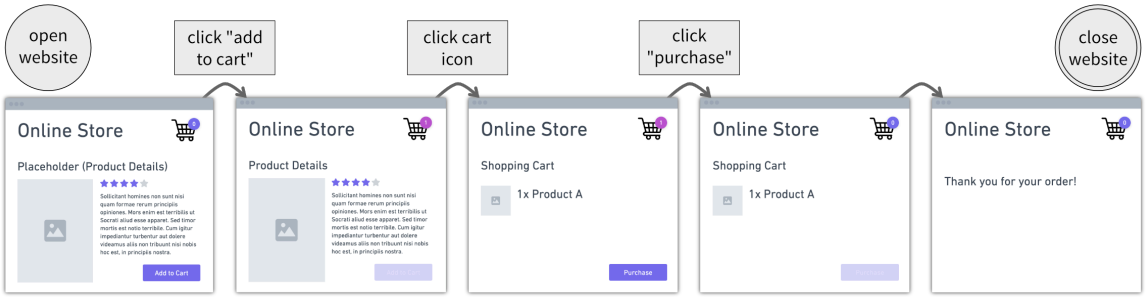


Figure 2.2.: A simple mockup illustrating the user experience flow on an e-commerce website, from opening the page and looking at a product to purchasing it.

A key part of this process is to **empathize with the user's experience**. How will users interact with the product? Under what conditions will they use it, and for how long? These factors introduce

constraints that must shape your design. For example, if a website is frequently accessed on smartphones with large text settings, the layout must remain functional and visually appealing under those conditions.

Validating your design with real users is essential. **Every design decision is a bet**—an assumption about what users need and how they'll respond. Before committing significant resources to its implementation, you need to test whether that bet is likely to pay off.

This makes UX design an **iterative process**: the design is refined through multiple cycles until it reasonably satisfies its intended goals. And because software is flexible—unlike physical products that are costly to alter after production—**you should continue testing and refining your product as you build it**. Use this adaptability to your advantage by iterating continuously, ensuring the product evolves with your users' needs over time.

2.2. Research Output: Data Analysis Results

No matter how much programming your project requires, nearly all scientific work involves analyzing your data to create **result plots for publications or presentations**—the final output of your code that will be seen by others.

When analyzing data, the process is typically divided into two phases:

1. **Exploratory analysis**: This involves generating a variety of plots to gain a deeper understanding of your data, such as identifying correlations between variables. It's often a quick and dirty process to help you familiarize yourself with the dataset.
2. **Explanatory analysis**: This focuses on creating refined, polished plots intended for communicating your findings to others, such as in a publication or presentation. These visuals are designed to clearly convey your results to an audience that may not be familiar with your data.

But before we dive into how to conduct exploratory and explanatory analyses, let's first take a quick look at what data actually is.

Data Types

In one form or another, you're research will rely on data, both collected or generated by yourself and possibly others.

Structured vs. Unstructured Data

Data can take many forms, but one key distinction is between structured and unstructured data (Figure 2.3).

Structured data is organized in rows and columns, like in Excel spreadsheets, CSV files, or relational databases. Each row represents a sample or observation (a data point), while each column corresponds to a variable or measurement (e.g., temperature, pressure, household income, number of children).

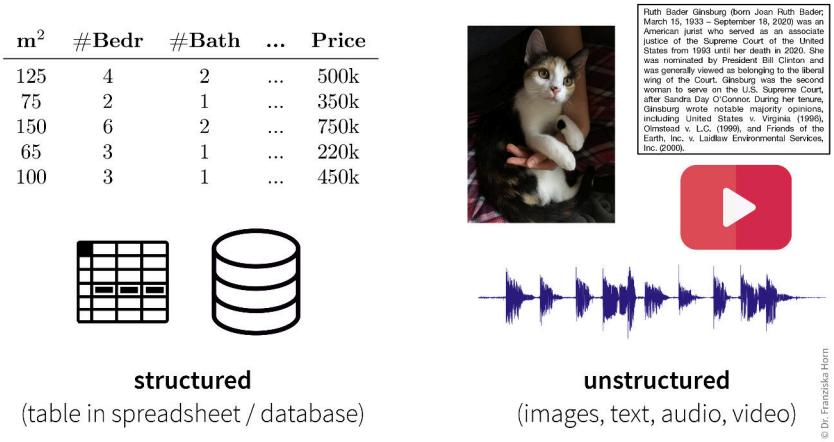


Figure 2.3.: Structured and unstructured data.

Unstructured data, in contrast, lacks a predefined structure. Examples include **images, text, audio recordings, and videos**, typically stored as separate files on a computer or in the cloud. While these files might include structured metadata (e.g., timestamps, camera settings), the data content itself can vary widely—for instance, audio recordings can range from seconds to hours in length.

Structured data is often *heterogeneous*, meaning it includes variables representing different kinds of information with distinct units or scales (e.g., temperature in °C and pressure in kPa). Unstructured data tends to be *homogeneous*; for example, there's no inherent difference between one pixel and the next in an image.

> **i** This book focuses on structured data
>
> Even though unstructured data is common in science (e.g., microscopy images), for simplicity, this book focuses on structured data. Furthermore, for now we'll assume that your data is stored in an Excel or CSV file, i.e., a spreadsheet with rows (samples) and columns (variables), on your computer. Later in Chapter 6, we'll discuss more advanced options for storing and accessing data, such as databases and APIs.

Programming Data Types

Each variable in your dataset (i.e., each column in your spreadsheet) is represented as a specific data type, such as:

- **Numbers** (integers for whole numbers or floats for decimals)
- **Strings** (text)
- **Boolean values** (true/false)

In programming, these are so-called *primitive data types* (as opposed to composite types, like lists, sets, or dictionaries containing multiple values, or user-defined objects) and define how information is stored in computer memory.

Data types in Python

```python
# integer
i = 42
# float
x = 4.1083
# string
s = "hello world!"
# boolean
b = False
```

Statistical Data Types

Even more important than how your data is stored, is understanding what your data *means*. Variables fall into two main categories:

1. **Continuous (numerical) variables** represent measurable values (e.g., temperature, height). These are usually stored as floats or integers.
2. **Discrete (categorical) variables** represent distinct options or groups (e.g., nationality, product type). These are often stored as strings, booleans, or sometimes integers.

Misleading data types

Be cautious: a variable that looks numerical (e.g., `1`, `2`, `3`) may actually represent categories. For example, a `material_type` column with values `1`, `2`, and `3` might correspond to *aluminum*, *copper*, and *steel*, respectively. In this case, the numbers are IDs, not quantities.

Recognizing whether a variable is continuous or discrete is crucial for creating meaningful visualizations and using appropriate statistical models.

Time Series Data

Another consideration is whether your data points are linked by time. Time series data often refers to numerical data collected over time, like temperature readings or sales numbers. These datasets are usually expected to exhibit **seasonal patterns or trends** over time.

However, **nearly all datasets involve some element of time**. For example, if your dataset consists of photos, timestamps might seem unimportant, but they could reveal trends—like changes in image quality due to new equipment.

Always record timestamps

Always include timestamps in your data or metadata to help identify potential correlations or unexpected trends over time.

Sometimes, you may be able to collect truly time-independent data (e.g., sending a survey to 1,000 people simultaneously and they all answer within the next 10 minutes). But usually, your data collection will take longer and external factors—like an election during a longer survey period—might unintentionally affect your results. By tracking time, you can assess and adjust for such influences.

Exploratory Analysis

In this initial analysis, the goal is to get acquainted with the data, check if the trends and relationships you anticipated are present, and uncover any unexpected patterns or insights.

- Examine the **raw data**:
  - Is the dataset complete, i.e., does it contain all the variables and samples you expected?
- Examine **summary statistics** (e.g., mean, standard deviation (std), min/max values, missing value count, etc.):
  - What does each variable mean? Given your understanding of the variable, are its values in a reasonable range?
  - Are missing values encoded as NaN (Not a Number) or as 'unrealistic' numeric values (e.g., -1 while normal values are between 0 and 100)?
  - Are missing values random or systematic (e.g., specific measurements might only be collected under certain conditions; in a survey rich people are less likely to answer questions about their income)? This can influence how missing values should be handled, e.g., whether it makes sense to impute them with the mean or some other specific value (e.g., zero).
- Examine the **distributions of individual (continuous) variables**:

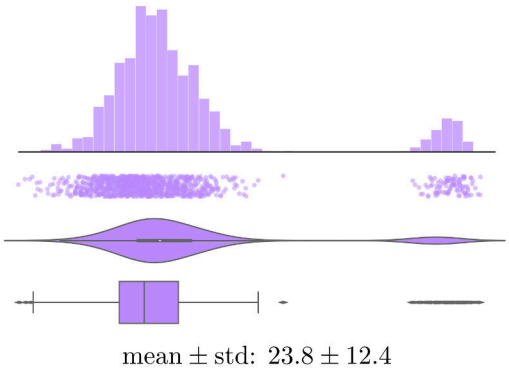


Figure 2.4.: Histogram, strip plot, violin plot, box plot, and summary statistics of the same values.

  - Are there any outliers? Are these genuine edge cases or can they be ignored (e.g., due to measurement errors or wrongly encoded data)?
  - Is the data normally distributed or does the plot show multiple peaks? Is this expected?
- Examine **trends over time** (by plotting variables over time, even if you don't think your data has a meaningful time component, e.g., by lining up representative images according to their timestamps to see if there is a pattern):

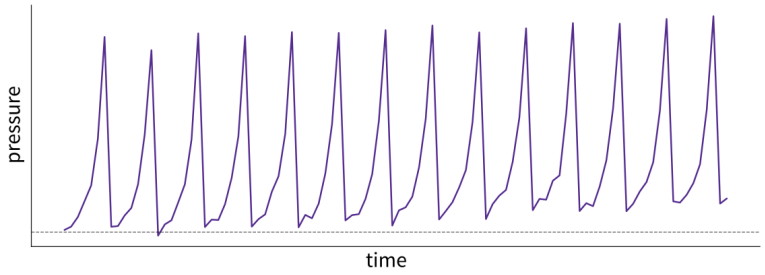


Figure 2.5.: This plot shows fictitious data of the pressure in a pipe affected by fouling—that is, a buildup of unwanted material on the pipe's surface, leading to increased pressure. The pipe is cleaned at regular intervals, causing a drop in pressure. However, because the cleaning process is imperfect, the baseline pressure gradually shifts upward over time.

  - Are there time periods where the data was sampled irregularly or samples are missing? Why?
  - Are there any (gradual or sudden) data drifts over time? Are these genuine changes (e.g., due to changes in the raw materials used in the process) or artifacts (e.g., due to a malfunctioning sensor recording wrong values)?

- Examine **relationships between two variables**:

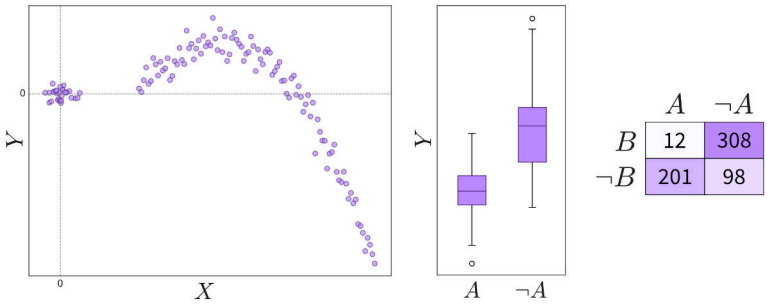


Figure 2.6.: Depending on the variables' types (continuous or discrete), relationships can be shown in scatter plots, box plots, or a table. Please note that not all interesting relations between the two variables can be detected through a high correlation coefficient, so you should always check the scatter plot for details.

  - Are the observed correlations between variables expected?

- Examine **patterns in multidimensional data** (using a parallel coordinate plot):

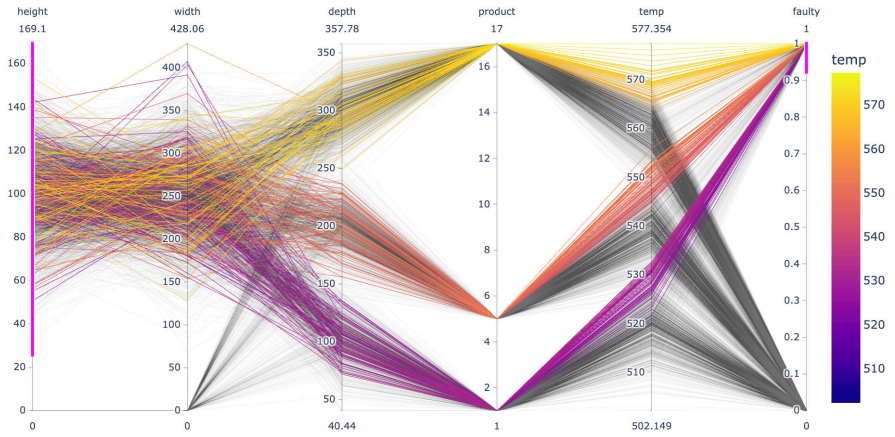


Figure 2.7.: Each line in a parallel coordinate plot represents one data point, with the corresponding values for the different variables marked at the respective y-axis. The screenshot here shows an interactive plot created using the Python `plotly` library. By selecting value ranges for the different dimensions (indicated by the pink stripes), it is possible to spot interesting patterns resulting from a combination of values across multiple variables.

- Do the observed patterns in the data match your understanding of the problem and dataset?

Explanatory Analysis

Most of the plots you create during an exploratory analysis are likely for your eyes only. Any plots you do choose to share with a broader audience—such as in a paper or presentation—should be refined to **clearly communicate your findings**. Since your audience is much less familiar with the data and likely lacks the time or interest to explore it in depth, it's essential to make your results more accessible. This process is often referred to as *explanatory analysis* [20].

> ⚠ Don't force an exploratory analysis onto your audience
>
> Don't "just show all the data" and hope that your audience will make something of it—understand what they need to answer the questions they have and tailor your results accordingly.

When choosing and designing your plots, keep the user experience in mind. Ask yourself: *Does this visualization clearly convey the results? Is it easy to understand and interpret?*
Use the following steps to help you evaluate and improve your plot design.

Step 1: Choose the Right Plot Type

- Get inspired by visualization libraries (e.g., here or here), but avoid the urge to create fancy graphics; sticking with common visualizations makes it easier for the audience to correctly decode the presented information.
- Don't use 3D effects!
- Avoid pie or donut charts (angles are hard to interpret).
- Use line plots for time series data.

- Use horizontal instead of vertical bar charts for audiences that read left to right.
- Start the y-axis at 0 for area & bar charts.
- Consider using small multiples or sparklines instead of cramming too much into a single chart.

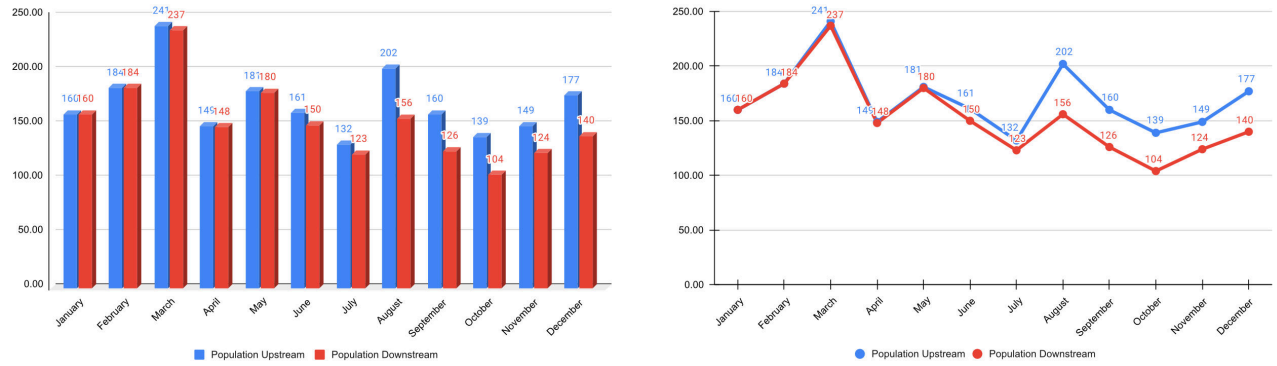


Figure 2.8.: *Left:* Bar charts (especially in 3D) make it hard to compare numbers over a longer period of time. *Right:* Trends over time can be more easily detected in line charts. [Example adapted from: *Storytelling with Data* by Cole Nussbaum Knaflic]

Step 2: Cut Clutter / Maximize Data-to-Ink Ratio

- Remove border.
- Remove gridlines.
- Remove data markers.
- Clean up axis labels.
- Label data directly.

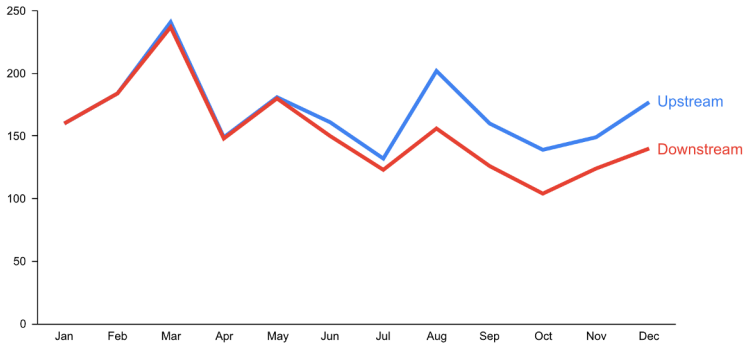


Figure 2.9.: Cut clutter! [Example adapted from: *Storytelling with Data* by Cole Nussbaum Knaflic]

Step 3: Focus Attention

- Start with gray, i.e., push everything in the background.
- Use pre-attentive attributes like color strategically to highlight what's most important.

- Use data labels sparingly.

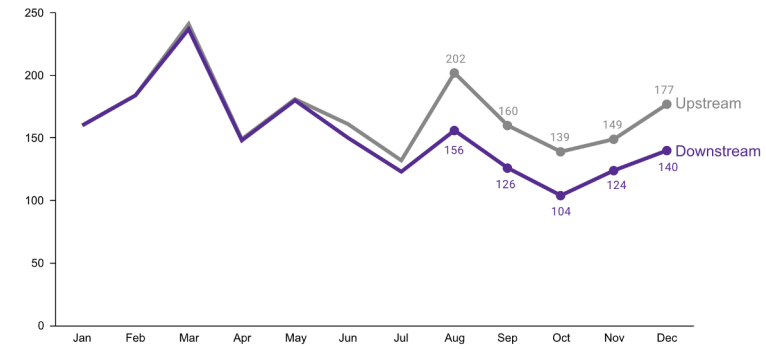


Figure 2.10.: Start with gray and use pre-attentive attributes strategically to focus the audience's attention. [Example adapted from: *Storytelling with Data* by Cole Nussbaum Knaflic]

Step 4: Make Data Accessible

- Add context: Which values are good (goal state), which are bad (alert threshold)? Should the value be compared to another variable (e.g., actual vs. forecast)?
- Leverage consistent colors when information is spread across multiple plots (e.g., data from a certain country is always shown in the same color).
- Annotate the plot with text explaining the main takeaways. If this is not possible, e.g., in interactive dashboards where the data keeps changing, the title can instead include the question that the plot should answer (e.g., "Is the material quality on target?").

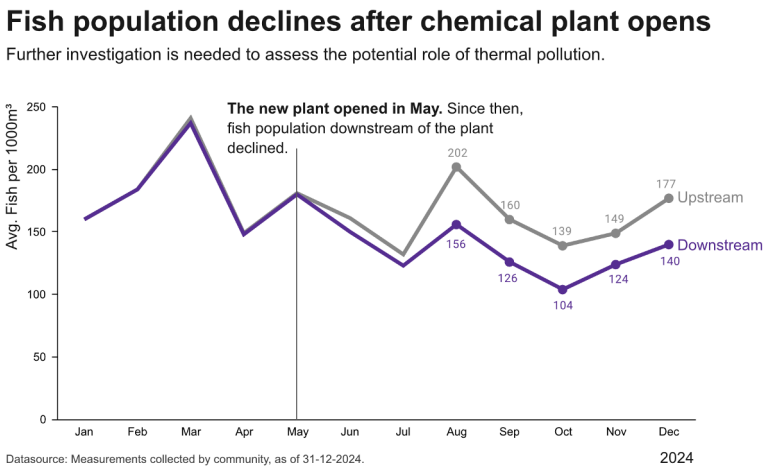


Figure 2.11.: Tell a story. [Example adapted from: *Storytelling with Data* by Cole Nussbaum Knaflic]

2.3. Draw Your Output

You may not have looked at your data yet—or maybe you haven't even collected it—but it's important to start with the end in mind.[1] Similar to how UX designers create mockups to visualize a product before it gets built, we also need to envision what the final output of our code should look like before writing it. The key difference is that, instead of users interacting directly with the software, they'll typically only see the results—such as plots or tables—produced by your script, often in a journal article or presentation.

Based on your takeaways from the previous chapter—about the problem you're solving and the metrics you should use to evaluate your solution—try sketching what your final results might look like. Put yourself in your audience's shoes and consider what they need to address their questions and concerns. Ask yourself: ***What figures or tables would best communicate the advantages of my approach?***[2]

Depending on your research goal, your results might be as simple as a single number, such as a p-value or the average response from the people you surveyed. However, if you're reading this, you're likely tackling something that requires a more complex analysis. For example, you might compare your model's overall performance to several baseline approaches or illustrate how your solution converges over time (Figure 2.12).

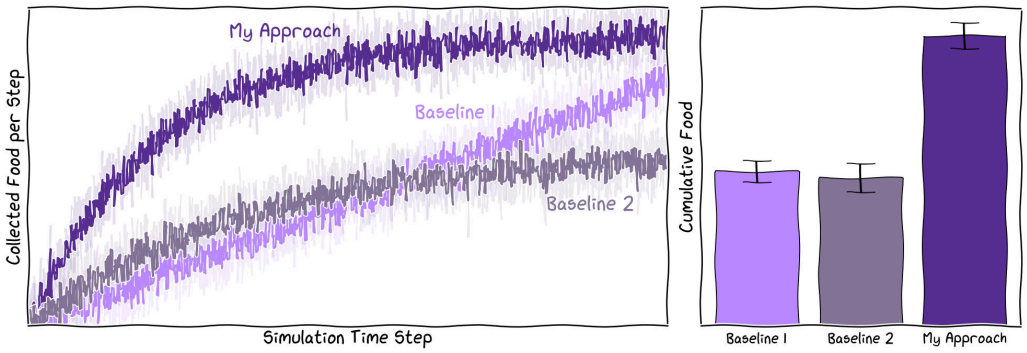


Figure 2.12.: Exemplary envisioned results: The plots show the outcome of a multi-agent simulation, where 'my approach' clearly outperforms two baseline methods. In this simulation, a group of agents is tasked with locating a food source in the environment and transporting the food back to their home base piece by piece. The ideal algorithm identifies the shortest path to the food source quickly to maximize food collection. Each algorithmic approach is tested 10 times using different random seeds to evaluate reliability. The plots display the mean and standard deviations across these runs. *Left:* How quickly each algorithm converges to the shortest path (resulting in the highest number of agents delivering food back to the home base per step). *Right:* Cumulative food collected by the end of the simulation.[3]

[1] A former master's student that I mentored humorously called this approach "plot-driven development," a nod to test-driven development (TDD) in software engineering, where you write a test for your function first and then implement the function to pass the test. Plot-driven development later turned into clarity-driven development, but the idea behind it stayed the same: understand what output your software should create before writing the code to make it happen.

[2] If you're already drafting a paper or presentation, you could even use these sketches of your results as placeholders to make sure they are advancing the story you're trying to tell.

Just as UX designs are **tested with real users**, you should share your result sketches with others and observe where they struggle. Do your colleagues interpret your plots the way you intend? Are they understanding the message you're trying to convey?

It's important to remember that **your actual results might look very different** from your initial sketches—they might even show that your solution performs worse than the baseline! This is completely normal. The **scientific method is inherently iterative**, and unexpected results are often a stepping stone to deeper understanding. When your results deviate from your expectations, analyzing those differences can sharpen your intuition about the data and help you form better hypotheses in the future. But by starting with a clear plan, you can generate results more efficiently and quickly pivot to a new hypothesis if needed.

Before you continue

At this point, you should have a clear understanding of:

- The specific results (tables and figures) you want to create to show how your solution outperforms existing approaches (e.g., in terms of accuracy, speed, etc.).
- What types of data you're working with to produce these results (e.g., what kind of values will be stored in the rows and columns of your spreadsheet).

[3]These plots were generated with Python using matplotlib's `plt.xkcd()` setting and the xkcd script font. But a pen and paper sketch works just as well.

3. State & Flow: How?

Now that you know what outputs (e.g., result plots) you want to create, are you itching to start programming? Hold on for a moment!

One of the most **common missteps** I've seen junior developers take is **jumping straight into coding without first thinking through what they actually want to write**. Imagine trying to construct a house by just laying bricks without consulting an architect first—halfway through you'd probably realize the walls don't align, and you forgot the plumbing for the kitchen. You'd have to tear it down and start over! To avoid this fate for your software, it's essential to make a plan and sketch out the final design first (Figure 3.1).

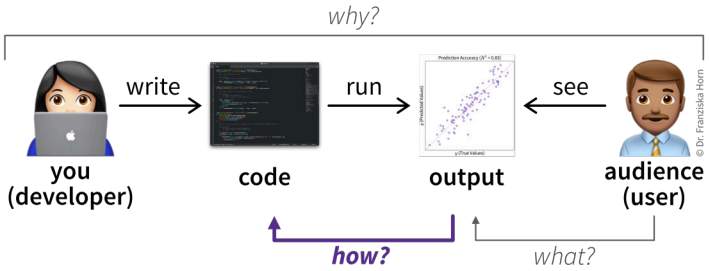


Figure 3.1.: Before we write code, we should first think about what it is we actually want to write, i.e., how this code is supposed to work.

Your software design doesn't have to be perfect—we don't want to overengineer our solution, especially since many details only become clear once we start coding and see how users interact with the software. But the more thought you put into planning, the smoother and faster execution will be.

Unlike a house, where your design will be quite literally set in stone, code should be designed with flexibility in mind. While expanding a house to add extra rooms for a growing family—and then removing them again when downsizing for retirement—would be costly and difficult, this kind of adaptability is exactly what we strive for in software. Our goal is to **create code that can evolve with changing requirements**.

To make sure your designs will be worthy of implementation, this chapter introduces key paradigms and best practices that will help you create **clean, maintainable code that's easy to extend and reuse** in future projects.

Difference Between Commercial Software and Research Code?

Commercial software applications and the scripts used in your research share many of the same core principles—both involve writing code to produce outputs, and both benefit from the same standards of good code quality. That's the topic of this chapter.

However, full-scale software applications, especially those with graphical interfaces, are naturally more complex than a script that generates static results. The additional components required for production-grade software are covered later in Chapter 6.

3.1. Working Code

Before we talk about the best practices for creating *good code*, let's first examine what our code is actually supposed to do and the steps we need to take to reach our goal—creating *working code*.

Impact Beyond Code

A program would be useless if it did some computation but never revealed the results to anyone. In the end, we always want to produce some kind of **effect on the world outside of our code** by interacting with users, data stores, or other external systems (Figure 3.2).

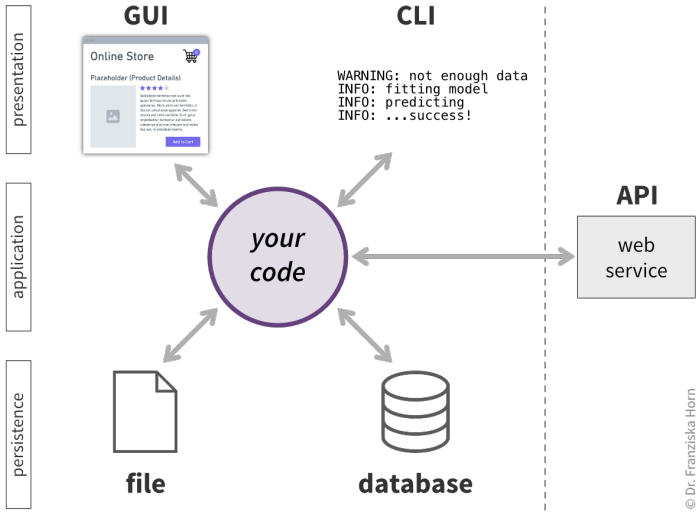


Figure 3.2.: To be useful, our code needs to have an effect on the outside world. First and foremost, a program interacts with its users through a **Graphical User Interface (GUI)** or **Command Line Interface (CLI)** where outputs are presented to users and they may be prompted for inputs. Next, our program reads and writes data from and to **data stores** like **files** or **databases**. These may contain inputs necessary for calculations or be used to persist (i.e., save) user inputs or the results from our calculations. Finally, our program may also interact with an external system, such as a web service, through an **API (Application Programming Interface)**, an interface designed specifically for two software components to communicate and exchange data. These external systems in turn might also have their own user interfaces and data stores, but these are of no concern for our program.

The impact of our code can take many different forms:

- Writing a log message with the result from a calculation to the console.

- Displaying information in a graphical user interface (GUI) and asking the user for input.
- Reading inputs from or writing outputs to a file (e.g., an Excel spreadsheet or a PNG image).
- Creating, reading, updating, or deleting records in a database.
- Querying an external web API (e.g., through a GET or POST request).
- Sending an email.
- Transmitting the necessary signals to move the actuators in a robotic arm.

In the previous chapter, we've talked at length about what outputs would be most beneficial for our users and how we can create an optimal user experience. In Chapter 6, we'll cover how our program can interact with external systems through APIs. So let's now turn to the topic of data persistence.

Data Persistence

As our code runs, all the data it generates through calculations is only present in the computer's working memory (RAM). Once the program terminates—whether because it is finished with its task or crashes due to an error—these in-memory values are gone. It is therefore often helpful to **persist (i.e., save) intermediate results** while our code is executed.

Especially when **different scripts or processes create and consume these intermediate results**, they should be stored in a **format that makes sense for both sides**. For example:

- A user fills out a form on a website. Their information is stored in a database so it can later be retrieved and displayed on their account page.
- You run simulations that output data to CSV files. These files are then used by a separate script to generate plots.
- You define model settings in a `config.json` file, which your script reads to initialize the model with the correct configuration.

It's important to **carefully design the structure** of this intermediate data—what fields it includes, what they're called, and how they're organized. Because if you later need to **change the format**, you'll have to **update both the producer and consumer processes**, and potentially **migrate existing data**.

To design an effective structure, start by identifying the fields required by the downstream process. Then consider whether other processes will use this data and if they might need additional fields. Don't overengineer it—we're not trying to future-proof against every possible scenario. But it's worth considering whether to store data at a finer level of detail than what's currently needed. It's much easier to extract or summarize detailed data later than to reverse-engineer missing details from aggregate values. For example, if you want to plot how values change over time, you'll need to record variables at every time step—not just the final outcome of a simulation (duh!).

> **Plan your experiments**
>
> If your code is supposed to do more than generate result plots—for example, if you're running a simulation—you also need to **plan your experiment runs**.
> Think about the data each experiment run will generate and how you'll process it later. For instance, you might need a separate script that loads data from different runs to create additional plots comparing the performance of multiple models. This is simplified by using **a clear naming**

convention for all output files, such as:

`{dataset}_{model}_{model_settings}_{seed}.csv`

To determine all possible configuration variants for your experiments, consider:

- The models or algorithms you want to compare (your approach and relevant baselines).
- The hyperparameter settings you want to test for each model.
- The datasets or simulation environments on which you'll evaluate your models.
- If your setup includes randomness (e.g., model initialization or simulation stochasticity), how many different random seeds you'll use to estimate variability.

Ideally, your code should allow you to **run the same script with different parameter configurations**, making it easy to test multiple setups. If your experiments will take several days, check whether you can run them on a compute cluster, submitting each experiment as a separate job to **run in parallel**.

State & Flow

While our final program needs to have an effect on the outside world, the code itself is, at its core, about **transforming inputs into outputs**. For example, your script might contain the steps necessary to turn the data read from a spreadsheet (input) into a result plot shown to the user (output).

Our code defines the sequence of operations (e.g., adding or multiplying values), conditional statements (e.g., `if/else`), and loops (e.g., `for` and `while`) that drive this transformation process—collectively known as an algorithm or **control flow**.
As these steps are executed, both the inputs and the intermediate results are stored in variables, which represent the current **state** of the program.

Data Structures

Variables (like `y`) store values (like `42`). When we define a variable, it points to a location in the computer's memory—a sequence of bits (0s and 1s)—that can be decoded into a specific value. To decode these bits correctly, the system must know the **data type** of the value being stored. For example, the binary `101010` represents `42` as an integer, but if interpreted as an ASCII character, it yields `"*"`. These data types range in complexity:

- **Primitive data types**: Simple values like integers or strings (as discussed in Section 2.2).
- **Composite data types**: Structures like lists, sets, and dictionaries that hold multiple values.
- **Nonlinear data types**: Structures like trees and graphs, used to model complex relationships such as hierarchies or networks.
- **User-defined objects**: Custom structures defined in classes, tailored for specific use cases.

```
# primitive data type: float
x = 4.1083

# composite data type: list
my_list = ["hello", 42, x]

# composite data type: dict
my_dict = {
    "key1": "hello",
    "key2": 42,
    "key3": my_list,
}
```

Depending on the problem you're solving, **choosing the right data structure** can make a big difference. For example, structures like trees or graphs can simplify operations such as finding related values or navigating nested relationships. That's why it's important to consider not just the sequence of operations in our code, but also how the data is structured.

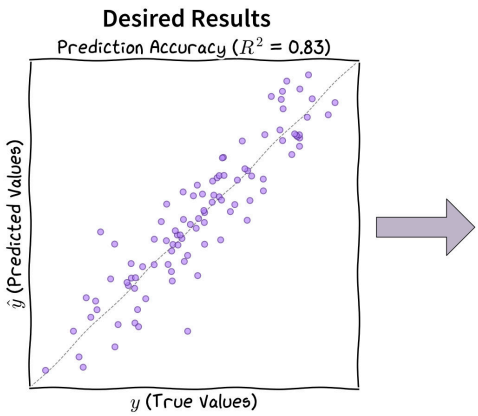
We can define custom data structures using **classes**. These are especially useful when multiple values represent a single entity. For instance, instead of managing separate variables like `first_name`, `last_name`, and `date_of_birth`, we can group them into a single **object** of a `Person` class.

While dictionaries could also store related fields, they offer less control: keys and values can be added or removed dynamically and without constraints. In contrast, classes provide structure and validation, such as enforcing required attributes during object creation or controlling how attributes are accessed or updated via class methods.

```
from datetime import date

class Person:
    def __init__(self, first_name: str, last_name: str, date_of_birth: date):
        self.first_name = first_name
        self.last_name = last_name
        if date_of_birth > date.today():
            raise ValueError("Date of birth cannot be in the future.")
        # this attribute is private (by prefixing _) so that it is not accessed outside of the cl
        self._date_of_birth = date_of_birth

    def get_age(self) -> int:
        # code outside the class only gets access to the person's age
        today = date.today()
        age = today.year - self._date_of_birth.year
        if (today.month, today.day) < (self._date_of_birth.month, self._date_of_birth.day):
            age -= 1
        return age

if __name__ == '__main__':
   new_person = Person("Jane", "Doe", date(1988, 5, 20))
```

Step by Step: From Output to Input

To determine the sequence of instructions for obtaining our desired outputs, we can **work backward to figure out which inputs we need and how these should be transformed** (Figure 3.3).



Required Data

	x1	x2	x3	x4	x5	y	y_pred
0	0.154	0.385	0.790	0.526	0.672	1.388	1.547
1	0.740	0.596	0.982	0.242	0.825	2.109	2.381
2	0.263	0.611	0.142	0.456	0.816	2.741	2.781
3	0.534	0.600	0.026	0.116	0.560	2.361	1.390
4	0.015	0.312	0.746	0.600	0.560	0.167	0.409
5	0.919	0.070	0.617	0.837	0.858	3.283	2.715
6	0.901	0.801	0.099	0.912	0.206	5.134	5.623
7	0.033	0.911	0.522	0.951	0.716	1.592	1.583
8	0.957	0.195	0.507	0.582	0.816	3.243	3.241
9	0.137	0.211	0.286	0.515	0.119	0.265	-0.046
10	0.284	0.377	0.248	0.777	0.849	1.813	2.652
11	0.606	0.403	0.180	0.672	0.125	3.015	2.643
12	0.944	0.885	0.584	0.164	0.054	3.836	4.500

Figure 3.3.: Work backward from the desired results to determine what data is needed to create them.

Let's map out the steps required to create the above scatter plot displaying actual values y (`y`), model predictions $\hat{y}$ (`y_pred`), and the R^2 value (`R2`) in the title to indicate the model's goodness of fit:

1. To create the plot, we need `R2`, `y`, and `y_pred`.
   ```
   plot_results(R2, y, y_pred)
   ```
2. `R2` can be computed from the values stored in `y` and `y_pred`.
   ```
   R2 = compute_r2(y, y_pred)
   ```
3. `y` can be loaded from a file containing test data.
   ```
   y = ...
   ```
4. `y_pred` must be estimated using our model, which requires:
   - The corresponding input values (`x1` … `x5`) from the test data file.
   - A trained model that can make predictions.

   ```
   X_test = ...
   y_pred = model.predict(X_test)
   ```
5. To obtain a trained model, we need to:
   - Create an instance of the model with the correct configuration.
   - Load the training data.
   - Train the model on the training data.

```
model = MyModel(config)
X_train, y_train = ...
model.fit(X_train, y_train)
```

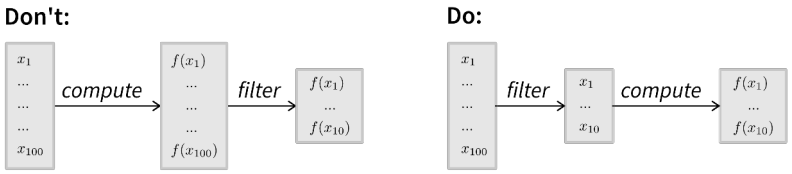
6. The model configuration needs to be provided by the user when running the script.

```
config = ...
```

Of course, in the actual implementation, each step will require further details (e.g., the formula for computing R^2 or how to load the training and test datasets). But since we know that following this sequence in reverse order will produce the desired output, this already provides us with the rough outline of our code (see Section 3.3 for how these steps could come together in the final script).

Optimize the ordering of steps

Some steps depend on others (e.g., we must fit our model before making predictions), but others can be performed in any order. Optimizing the sequence can **improve performance and efficiency**.
For example, if you're baking bread, you wouldn't preheat the oven hours before the dough has risen—that would waste energy. Similarly, when processing a large dataset, where you need to perform an expensive computation on each item but only a specific subset of these items is included in the final results, then it may be more efficient to filter the items first so you only compute values for the necessary items:



Designing an Algorithm

Some problems require more **creativity to come up with an efficient algorithm** that produces the correct output from a given input. Take the **traveling salesman problem**, where the goal is to find the shortest possible route through a given set of cities. Designing an efficient solution often involves choosing the right **data structures** (e.g., representing cities as nodes in a graph), using **heuristics**, and **breaking the problem down into simpler subproblems** (divide & conquer strategy). If you're working on a novel problem, studying algorithm design can be invaluable (one of my favorite books on the topic is *The Algorithm Design Manual* by Steven Skiena [31]).

However, for many common tasks, you can **leverage existing algorithms** instead of reinventing the wheel. For the rest of this book, we'll assume you already have a general idea of the steps your code needs to perform and focus on how to implement them effectively.

> **Practice algorithmic thinking**
>
> Coming up with efficient algorithms—even for simple problems—often takes practice. Platforms like LeetCode provide a fun way to build that skill by working through small, focused problems.

While the sequence of steps we derived above enables us to create *working code*, unfortunately, this is not the same as *good code*. A **script that consists of a long list of instructions** can be difficult to read, understand, and maintain. Since you'll likely be working with the same code for a while—and may even want to reuse parts of it in future projects—it's worth investing in a better design. Let's explore how to make that happen.

3.2. Good Code

When we talk about "good code" in this book, we focus on three key properties, each building upon and influencing the others:

1. **Easy to understand:** To work with code, you need to be able to understand it. When using a common library, it may be enough to know what a function does and how to use it, trusting that it works correctly under the hood. But with your own code, you also need to understand the implementation details. Otherwise, you'll hesitate to make changes and won't be able to confidently say that it behaves as expected. Since code is read far more often than it is written, making it easy to understand ultimately saves time in the long run.

2. **Easy to change:** Code is never truly finished. It requires ongoing maintenance—fixing bugs, updating dependencies (e.g., when a library releases a security patch with breaking changes), and adapting to evolving requirements, such as adding new features to stay competitive. Code that is easy to modify makes all of this much smoother.

3. **Easy to build upon and reuse:** Ideally, adding new features shouldn't mean proportional growth in your codebase. Instead, you should be able to reuse existing functionality. Following the DRY (Don't Repeat Yourself) principle—avoiding redundant logic—also makes code easier to maintain because updates only need to be made in one place.

These qualities (along with others, like being easy to test) can be achieved through **encapsulation**, i.e., breaking the code into smaller units, and ensuring that these units are **decoupled** and **reusable**. This approach has multiple benefits:

- Smaller units fit comfortably into your working memory, making them easier to understand and reason about, which lowers the overall cognitive load of your code.
- Since units are independent, you can change one without unintended side effects elsewhere.
- Well-designed, general-purpose building blocks allow you to quickly assemble more complex functionality—like constructing something out of LEGO bricks.

Analogy: The Cupcake Recipe

Before diving into how to write good code, let's explore some of the basic principles using a more relatable example: a cupcake recipe (Figure 3.4). Because who wants result plots when you could have cupcakes?!

Figure 3.4.: A chef writes a recipe, which then needs to be executed (baked) to generate the output (cupcakes) that the consumers enjoy.

Like code, a recipe is just **text that describes a sequence of steps to achieve a goal**. What you ultimately want are delicious cupcakes (your result plots). The recipe (your script) details how to transform raw ingredients (input) into the baked goods (output). But the steps don't mean anything unless you actually **execute them**—you have to bake the cupcakes (run `python script.py`) to get results.

So, how can we write a good recipe?

Breaking Down the Steps

To start, we **brainstorm all the necessary steps** for making cupcakes (Figure 3.5). For this, we can again **work backward from the final result** (cupcakes) through the intermediate steps (cake batter and frosting) until we reach the raw ingredients.

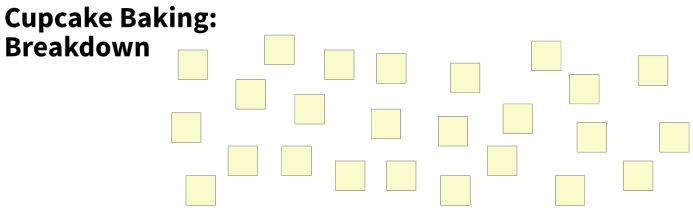


Figure 3.5.: Everything you need to make cupcakes. The unstructured brain dump we'll transform into a proper recipe.

You'll notice that your list includes both **ingredients** (in code: **variables containing data**) and **instructions for transforming those ingredients**, like melting butter or mixing multiple ingredients (in code: the **control flow**, i.e., statements, including conditionals (`if/else`) and loops (`for`, `while`), that create intermediate variables). These steps also need to follow a **specific order**—you wouldn't frost a cupcake before baking it! Often, steps **depend on one another**, requiring a structured sequence (Figure 3.6).

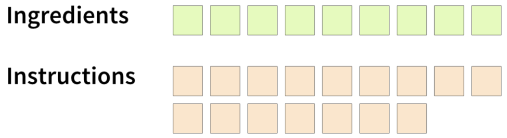


Figure 3.6.: A recipe usually includes a list of ingredients followed by instructions on what to do with them. The order matters, especially when one step depends on the completion of another.

Avoiding Repetition: Reusable Steps

If your recipe is part of a cookbook with multiple related recipes, some steps might apply to several of them. Instead of repeating those steps in every recipe, you can **group and document them in a separate section** (in code: define a **function**) and reference them when needed (Figure 3.7).

This follows the **Don't Repeat Yourself (DRY)** principle [32]. Not only does this reduce redundancy, but it also makes updates easier—if a general step changes (e.g., you refine a technique or fix a typo), you only need to update it in one place (**a single source of truth**).

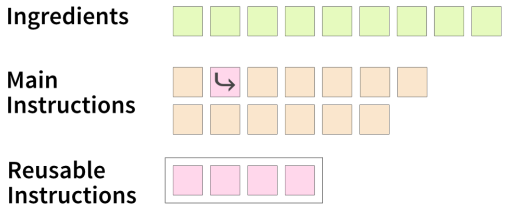


Figure 3.7.: Some instructions might be relevant in multiple recipes (e.g., general baking tips); instead of repeating them in every recipe, you can just refer to the page where they are described.

Organizing by Components

Looking at your recipe, you might notice that some **ingredients and instructions naturally group into self-contained components** (in code: **classes**). Structuring the recipe this way makes it clearer and easier to follow (Figure 3.8). Instead of juggling all the steps at once, you can **focus on creating one component at a time**—first the plain cupcake, then the frosting, then assembling everything.

This modular approach also allows **delegation**—for example, you could buy pre-made cupcakes and just focus on making the frosting. In programming, this means that **the final code doesn't need to know how each component was made—it only cares that the components exist**.

Making Variants with Minimal Effort

Organizing a recipe into components also makes it easier to **create variations** (Figure 3.9).

Two key concepts make this possible:

1. **Polymorphism**—each variation should behave like the original so it can be used interchangeably (in code: implement the same **interface**).

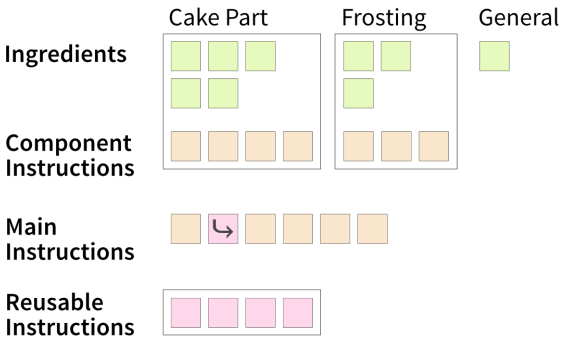


Figure 3.8.: For more complex recipes where the final dish consists of multiple components (like cupcakes, which have a cake part and frosting), grouping by component improves clarity.

2. **Code reuse**—instead of rewriting everything from scratch, **extend** the original recipe by specifying only the changes (in code: use **inheritance** or **mixins**).

For example, a chocolate frosting variant extends the plain frosting recipe by adding cocoa powder. The rest of the instructions remain unchanged.

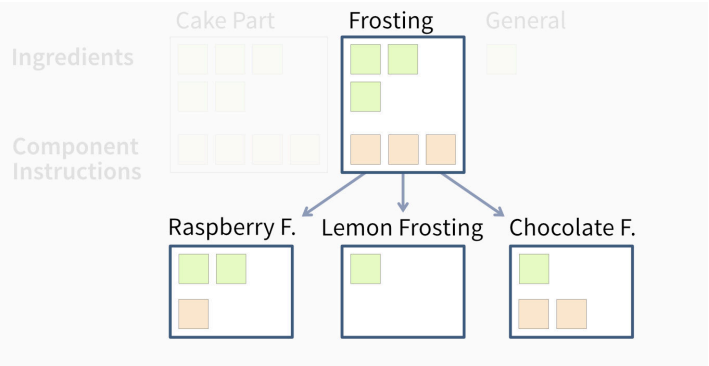


Figure 3.9.: Some components have variants that can be used as drop-in replacements. Since these variants extend the original (e.g., chocolate frosting builds on plain frosting), we reuse existing instructions and only describe what's new.

By applying these strategies, we turn a random assortment of ingredients and instructions into a well-structured, easy-to-follow recipe. This makes the cookbook not only clearer but the recipes also easier to maintain and extend—changes to general instructions only need to be made once, and readers can effortlessly create variations by reusing existing steps.

Now, let's take the same approach with code.

Encapsulated

Reading through a long script can be overwhelming. To make code easier to understand, we can **group related lines into units such as functions and classes**—a practice called **encapsulation**.

As a first step, **reorder the lines** in your code so they **follow overarching logical steps**, such as *data preprocessing* or *model fitting*. For example, if you define a variable at the top of the script, then execute some unrelated computations, and only later use that variable, move the definition down so it's closer to where it's actually used. This alone already makes your script easier to follow because the reader doesn't have to keep unrelated variables in mind while trying to understand what's happening. You'll know you've succeeded when your code is organized into separate blocks, each representing a distinct process step. You can even add a brief comment above each block describing, at a high level, what the following lines do.

```
# Before:
df = ...     # data
model = ...  # model

df_processed = ... # do something with df
model_fitted = ... # do something with model

# After:
### Step 1: data preprocessing
df = ...
df_processed = ...

### Step 2: model fitting
model = ...
model_fitted = ...
```

The next step is to **move these blocks into dedicated functions** (e.g., `preprocessing`) **or classes** (e.g., `Model`). Done well, this makes your code even more readable: your main script is reduced to a few lines of calls to these units, while the individual functions and classes can be understood in isolation. At that point, you won't even need the separating comments—your structure will speak for itself.

In the following sections, we'll explore how to use encapsulation effectively to reduce complexity.

From Complex to Complicated

A well-known software engineering principle is **KISS**—"Keep It Simple, Stupid!" While this is good advice at the level of individual functions or classes, it's unrealistic to expect an entire software system to be *simple*. Most real-world software consists of multiple interacting components, making it inherently *complicated* rather than simple. The goal is to avoid unnecessary **complexity** (Figure 3.10).

A **complex** system has many interconnected elements with dependencies that are difficult to trace. Making a small change in one place can unexpectedly break something elsewhere. This makes debugging and maintenance a nightmare.

[1]The fourth category is **chaotic**, where cause and effect are unknowable. In software, this could be compared to rare cosmic-ray-induced bit flips that cause random, unpredictable behavior.

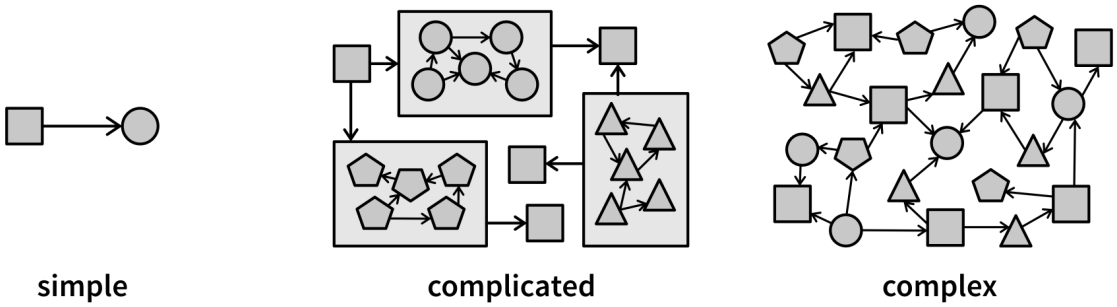


Figure 3.10.: In accordance with the Cynefin framework, (software) systems can have different levels of complexity [17]. A script that sequentially executes a few steps might be simple. Most software, however, is at least complicated—it consists of multiple interacting components but can still be broken down into understandable subsystems. A complex system, often referred to as "spaghetti code" or a "big ball of mud" [7], has so many interdependencies that its behavior is very difficult to predict—changes in one part can have unintended consequences elsewhere.[1] (Figure adapted from [24])

Instead, we aim for a **complicated** system—one that may have many components, but is structured into independent, well-organized subsystems [17]. This way, we can understand and modify individual parts without having to fully grasp the entire system.

Decomposing a system this way will always constitute a trade-off, requiring us to balance **local vs. global complexity** [17]. Consider two extremes:

- A system with **two massive 500-line functions** has low global complexity (only two things to keep track of) but high local complexity (each function is overwhelming to understand).
- A system with **500 tiny 2-line functions** has low local complexity (each function is simple) but high global complexity (understanding how they interact becomes difficult).

What constitutes a **"right-sized" unit**—small enough to be understandable, yet large enough to avoid excessive fragmentation—will depend on the context and your personal preferences (I usually aim for functions with around 5-15 lines of code).

i Cohesive units have a single responsibility

Following the **single responsibility principle**, each unit should be **cohesive**—focused on a single, well-defined task, with all of its internal code serving that purpose.
For example, a cohesive function performs a single task. If a boolean flag in its arguments determines which code path it follows and leads to different return values, it's better to split it into two separate functions.
Similarly, a cohesive class is designed to represent one specific concept completely—and only that concept. If a class contains many attributes, and one group of methods uses one subset of those attributes while another group relies on a different, non-overlapping subset, this suggests the class no longer adheres to the single responsibility principle and should be broken into smaller, more focused classes.
However, the reverse is also true: units can be **too narrowly defined to represent a complete task**. For example, if three functions must always run in a specific order to produce a usable

result, and none is ever called independently, they should likely be combined into a single function.

Interface vs. Implementation

The key idea of encapsulation is to establish a clear **boundary** between a unit's internal logic and the rest of the program. This allows us to hide the complex **implementation** (the underlying code that users of the unit don't need to see) behind a simple **interface** (the part exposed for use). This creates an effective **abstraction**, reducing cognitive load: users only need to understand how to interact with the unit, not how it works internally.

The **interface** serves as a contract, promising consistent behavior over time. The **implementation**, in contrast, is internal and can change freely—so others shouldn't depend on it.

Let's look at a simple example to see this in action:

```
def n_times_x(x, n):
    result = 0
    for i in range(n):
        result += x
    return result

if __name__ == '__main__':
    # call our function with some values
    my_result = n_times_x(3, 5)
    print(my_result)  # should output 15
```

The **interface** (also called signature) of the `n_times_x` function consists of:

- The function name (`n_times_x`)
- The input arguments (`x` and `n`)
- The return value (`result`)

> Make interfaces explicit
>
> Interfaces should be **explicit and foolproof**, especially since not all users read documentation carefully.
> For example, if a function relies on **positional arguments**, users might accidentally swap values, leading to hard-to-find bugs. Using **keyword arguments** (i.e., forcing users to specify argument names) makes the interface clearer and reduces errors.

As long as the interface remains unchanged, we can freely **modify the implementation**. For instance, we can replace the inefficient loop with a proper multiplication:

```
def n_times_x(x, n):
    return n * x
```

This change improves efficiency, but since the function still works the same way externally (it's called with the same arguments and returns the same results), no updates are required in other parts of the code. This is the power of clear boundaries.

However, changing a function's **name**, for example, is another story. If we rename `n_times_x`, every reference to it must also be updated. Modern IDEs can automate this within a project, but if the function is used in external code (e.g., if it is part of an open source library), renaming requires a **deprecation process** to transition users gradually.

This is why **choosing stable, well-thought-out interfaces upfront saves effort in the long run**.

> Go deep
>
> Powerful units have **narrow interfaces with deep implementations**—they expose only what's necessary while handling meaningful computations internally [26]. For example:
>
> - A function might take just two inputs (narrow interface) but span ten lines of logic (deep implementation).
> - Or a one-liner function might implement a complex formula that users shouldn't need to understand.
>
> But if your function is called `n_times_x`, and the implementation is just a one-liner doing exactly that, the abstraction does not help to reduce cognitive load. As a general rule, **a unit's interface should be significantly easier to understand than its implementation.**

Class Interfaces

As we already discussed, a **class groups attributes and methods that belong to a single entity**. For well-structured classes, we should carefully control **what is public and what is private**:

- **Public attributes & methods** form the external interface—changing them may break code elsewhere.
- **Private attributes & methods** (meant for internal use) may be modified anytime.

It is often useful to **make attributes private** and then **control how they are accessed or modified through methods**. This also implies that any external function that directly accesses or updates an object's attributes may be better implemented as a method within the corresponding class.

> Public and private in different languages
>
> Access levels vary by programming language, for example:
>
> - **Java** has multiple levels (`public`, `protected`, `package`, `private`).
> - **Python** relies on convention rather than enforcement. Prefixing names with `_` or `__` signals they are **"private"**, though they remain accessible.
>
> While Python won't stop users from accessing private attributes, they do so **at their own risk**,

> knowing these details might change without notice.

Decoupled

> *Two components are connascent [i.e., coupled] if a change in one would require the other to be modified in order to maintain the overall correctness of the system.*
> – Meilir Page-Jones [27]

We do not want to just hide arbitrary implementation details behind an interface, but ideally these units should also be **decoupled**. This means they should have as **few dependencies as possible** on other parts of the code or external resources, especially if these may change frequently. This makes the code easier to understand since **each unit can be comprehended in isolation**. But even more importantly, it **simplifies modifications**: when requirements change or new features are needed, we want to minimize the number of places that require updates. In contrast, **if code is tightly coupled, a single change in one unit can ripple through multiple parts of the system**, increasing the likelihood of errors.

Make Coupling Explicit

To build a larger software system that does anything meaningful, we must compose it from multiple components. This inevitably means our units are coupled in some way—one function must call another, so they cannot exist in complete isolation. The goal is to **make this coupling explicit so that if one unit changes, it is clear how dependent units need to be updated**.

The most explicit form of coupling is referencing a function or class by name when calling it. Because the name is part of the unit's **public interface**, any change should include a deprecation warning to support a smooth transition. The more explicit the public interface—such as using keyword arguments instead of positional ones—the safer it is to use this unit, since compilers and static code analysis tools can detect changes or errors ahead of time, and IDEs can assist with refactoring.

Coupling becomes riskier when your code relies on a unit's **internal implementation details**, like private attributes, undocumented quirks, or unintended behaviors (e.g., a function producing odd results when given inputs outside its intended range). Often, multiple parts of the code must share assumptions, such as the meaning of certain values (e.g., the magic number `404` representing the HTTP status code "not found") or the use of a specific algorithm when encrypting and decrypting data.

The above examples are instances of static coupling, meaning they can be spotted in the source code directly. An even more error-prone form is **dynamic coupling**, visible only at runtime. For example, a class might require its methods to be called in a particular order to initialize all necessary values for a correct computation.

To minimize coupling risks, **identify the assumptions your code makes** about how and in which context it will be used. Don't just document the correct usage and hope it's followed, but make interactions explicit. This can mean defining formal interfaces, validating inputs, and enforcing usage constraints in code. Otherwise, the system becomes fragile, and even small changes in unrelated areas can cause breakages.

Layers of Abstraction

Ideally, our units should form a hierarchy with different **levels of abstraction** (Figure 3.11), where lower-level units can be used as building blocks to create more advanced functionality.

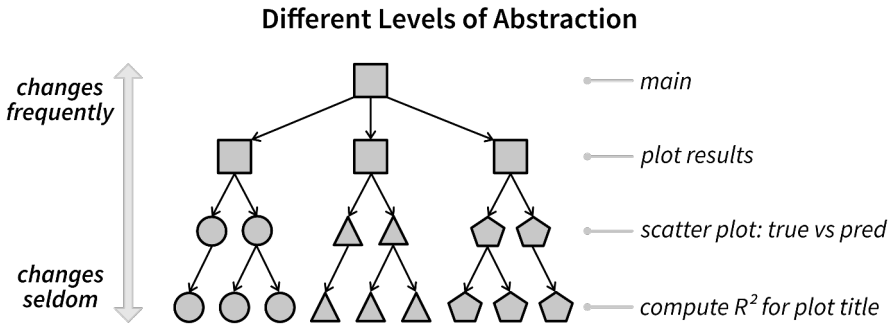


Figure 3.11.: Complicated systems in software design often follow a hierarchy of abstraction. For example, to plot the results of a predictive model you call a function to create a scatter plot of predicted vs. actual values, which internally calls another, more specific function to compute the R^2 value displayed in the plot's title.

At each level, you only need to understand **how to use** the functions at lower levels but **not their internals**. This **reduces cognitive load**, making it easier to navigate and extend the codebase with confidence.

However, it's important to note that **low-level functions should ideally be kept stable**, since everything built on top of them depends on their behavior. **If the interface of such a function changes, any code that relies on it will also need to be updated**. This applies not only to your own functions but also to external libraries your code depends on.

Standard libraries (i.e., those included with a programming language, not installed separately) are generally safe to build upon, as their functionality tends to be stable. In contrast, relying heavily on experimental or rapidly evolving libraries can lead to constant breakage as new versions introduce changes. To mitigate this, consider implementing an **anti-corruption layer**—a wrapper that provides a stable interface and handles necessary transformations. This centralizes adaptation and isolates the rest of your code from volatile dependencies.

Pure vs. Impure Functions

In addition to dependencies on code defined elsewhere, another form of coupling comes from **relying on external resources** (like files, databases, or APIs). These kinds of external dependencies are eliminated when our units are pure functions [25].

A **pure function** behaves like a mathematical function $f : x \to y$, taking inputs x and returning an output y—and given the same inputs, the results will be the same every time. **Impure functions**, on the other hand, have side effects—they interact with the outside world, such as reading from or writing to a file or modifying global state.

```python
def add(a, b):
    # pure: returns a computed result without side effects
    return a + b

def write_file(file_path, result):
    # impure: writes data to a file (side effect)
    with open(file_path, "w") as f:
        f.write(result)
```

Why favor pure functions?

- They are **predictable**—calling them with the same inputs always produces the same output.
- They are **easy to test** since they don't rely on external state.
- They **reduce unexpected behavior**, unlike impure functions, which may depend on changing external factors (e.g., the function may run fine the first time but then crash when trying to create a file that already exists or the computed results are suddenly different because in the meantime other code updated values stored in a database).

That said, impure functions are often necessary—code needs to have an effect on the outside world, otherwise why should we execute it in the first place? The trick is to **encapsulate as much essential logic as possible in pure functions** at the lower layers of abstraction, while isolating side effects in higher layers:

```python
def pure_function(data):
    # process data (pure logic)
    result = ...
    return result

def impure_function():
    with open("some_input.txt") as f:
        data = f.read()
    result = pure_function(data)
    with open("some_output.txt", "w") as f:
        f.write(result)
```

This structure ensures the core logic is **pure and testable**:

```python
# tests.py
def test_pure_function():
    data = "some test data"
    result = pure_function(data)
    assert result == "some test result"
```

Dependency Injection for Flexibility

To further decouple your code, use **dependency injection**: instead of hardcoding dependencies (e.g., file operations, database access), pass them as arguments. This allows functions to remain independent of specific implementations.

Without dependency injection:

```
class DataSource:
    def read(self):
        with open("some_input.txt") as f:
            return f.read()
    def write(self, result):
        with open("some_output.txt", "w") as f:
            f.write(result)

def fun_without_di():
    # create external dependency inside the function
    ds = DataSource()
    data = ds.read()
    result = ...
    ds.write(result)
```

With dependency injection:

```
def fun_with_di(ds):
    # dependency is passed as an argument
    data = ds.read()
    result = ...
    ds.write(result)

if __name__ == '__main__':
    ds = DataSource()
    fun_with_di(ds)
```

Now, `fun_with_di` only requires an object implementing `read` and `write`, but it **doesn't care** if it's a `DataSource` or something else that implements the same interface. This makes testing much easier:

```
# tests.py
class MockDataSource:
    def read(self):
        return "Mocked test data"
    def write(self, result):
        self.success = True

def test_fun_with_di():
    mock_ds = MockDataSource()
    fun_with_di(mock_ds)
    assert mock_ds.success
```

Dependency injection can also be **combined with pure functions** to make your code testable at different levels of abstraction.

Anti-Pattern: Global Variables

Besides external resources like files and databases, another source of **tight coupling and error-prone behavior** is relying on **global variables**. These are defined at the script level (often below import statements) and can be accessed or modified anywhere in the code.

Global variables can introduce **temporal coupling**, meaning the order of function execution suddenly matters. This can lead to subtle and hard-to-debug issues:

```
PI = 3.14159

def calc_area_impure(r):
    # since PI is not defined inside the function, the fallback is to use the global variable
    return PI * r**2

def change_pi(new_pi=3):
    # `global` is needed; otherwise, this would create a new local variable
    global PI
    PI = new_pi

if __name__ == '__main__':
    r = 5
    area_org = calc_area_impure(r)
    change_pi()
    area_new = calc_area_impure(r)
    assert area_org == area_new, "Unexpectedly different results!"
```

The safer approach is to pass values explicitly:

```
def calc_area_pure(r, pi):
    return pi * r**2
```

This avoids hidden dependencies and ensures reproducibility.[2]

To prevent unintended global variables, **wrap your script logic inside a function**:

```
def main():
   # main script code incl. variable definitions
   ...
```

[2] In some programming languages, you can define **constants**—variables whose values cannot be changed. These are often declared as global variables so that all functions can reference the same value. In Python, constants are defined by convention using `ALL_CAPS` variable names. However, Python does not enforce immutability, so the value can still be changed despite the naming convention.

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

This ensures that variables defined inside `main()` don't leak into the global namespace.

> Think you need global variables? Wrap your code inside a class instead!
>
> If multiple functions need to update the same set of variables, consider **grouping them into a class** instead of passing the same arguments repeatedly or relying on globals. A class can store shared state in attributes, accessed via `self`.

Call-By-Value vs. Call-By-Reference

Another common pitfall is **accidentally modifying function arguments**.

Depending on the programming language you use, input arguments are passed:

- **By value**: A copy of the data is passed.
- **By reference**: The function gets access to the original memory location and can modify the data directly.

Python uses **call-by-reference for mutable objects** (e.g., lists, dictionaries), which can lead to unexpected behavior:

```
def change_list(a_list):
    a_list[0] = 42  # modifies the original list
    return a_list

if __name__ == '__main__':
    my_list = [1, 2, 3]
    print(my_list)  # [1, 2, 3]
    new_list = change_list(my_list)
    print(new_list) # [42, 2, 3]
    print(my_list)  # [42, 2, 3]
```

To prevent such side effects, create a copy of the variable before modifying it. This ensures the original remains unchanged:

```
from copy import deepcopy

def change_list(a_list):
    a_list = deepcopy(a_list)
    a_list[0] = 42
    return a_list

if __name__ == '__main__':
```

```
    my_list = [1, 2, 3]
    new_list = change_list(my_list)
    print(my_list)  # [1, 2, 3]
```

To avoid sneaky bugs, **test your code to verify**:

1. **Inputs remain unchanged** after execution.
2. The function **produces the same output when called twice** with identical inputs.

Don't Repeat Yourself (DRY)

A more subtle form of coupling comes from violating the **DRY (Don't Repeat Yourself) principle**. For example, critical business logic—such as computing an evaluation metric, applying a discount in an e-commerce system, or validating data, like the rules that define an acceptable password—might be duplicated across several locations. If this logic needs to change (due to an error or evolving business requirements), you'd need to update multiple places, making your code harder to maintain and more prone to bugs. Make sure that important values and business rules are defined in a central place, the **single source of truth**, to avoid inconsistencies and duplication.

The most obvious DRY violation occurs when you catch yourself **copy-pasting code**—this is a clear signal to refactor it into a reusable function. However, DRY isn't just about code; it applies to **anything where a single change in requirements forces multiple updates**, including tests, documentation, or data stored across different databases [32]. To maintain your code effectively, aim to make each change in one place only. If that's not feasible, at least keep related elements close together—for example, documenting interfaces using docstrings in the code rather than in separate files.

Reusable

While we want to keep interfaces narrow and simple, we also want our units to be broadly applicable. This often means **replacing hardcoded values with function arguments**. Consider these two functions:

```
def a_plus_1(a):
    return a + 1

def a_plus_b(a, b=1):
    return a + b
```

The first function is highly specific, while the second is more general and adaptable. By providing **sensible default values** (`b=1`), we keep the function easy to use while allowing flexibility for more advanced cases.

> **Reusable code through refactoring, not overengineering**
>
> Reusable code isn't typically created on the first try because future needs are unpredictable. Instead of overengineering, focus on writing simple, clear code and adapt it as new opportunities for reuse emerge. This process, called *refactoring*, is covered in more detail in Section 5.7.

That said, if your function ends up with ten arguments, reconsider whether it has a **single responsibility**. If it's doing too much, breaking it into multiple functions is likely a better approach.

Polymorphism

When working with classes, reuse can be achieved through **polymorphism**, where multiple classes implement the same interface and can be used interchangeably. We've already applied this principle when we used `MockDataStorage` instead of `DataStorage` for testing. This approach can be useful in research code as well, for example, you could define a `Model` interface with `fit` and `predict` methods so multiple models can share the same experiment logic:

```
class Model:
    def fit(self, x, y):
        raise NotImplementedError
    def predict(self, x):
        raise NotImplementedError

class MyModel(Model):
    def fit(self, x, y):
        ...  # implementation
    def predict(self, x):
        y_pred = ...
        return y_pred

def run_experiment(model: Model):
    # this code works with any model that implements
    # appropriate fit and predict functions
    x_train, y_train = ...
    model.fit(x_train, y_train)
    x_test = ...
    y_test_pred = model.predict(x_test)

if __name__ == '__main__':
    # create a specific model (possibly based on arguments passed by the user)
    model = MyModel()
    run_experiment(model)
```

This way you can reuse your analysis code with all your models, which not only avoids redundancy, but ensures that all models are evaluated consistently.

Mixins

Another approach to reuse is using **mixins**—small reusable classes that provide additional functionality:

```python
import numpy as np

class ScoredModelMixin:
    def score(self, x, y):
        y_pred = self.predict(x)
        return np.mean((y - y_pred)**2)

class MyModel(Model, ScoredModelMixin):
    ...

if __name__ == '__main__':
    model = MyModel()
    # this uses the function implemented in ScoredModelMixin
    print(model.score(np.random.randn(5, 3), np.random.randn(5)))
```

Historically, deep class hierarchies with multiple levels of inheritance were common, but they often led to unnecessary complexity. Instead of forcing all functionality into a single class hierarchy, it's better to keep interfaces narrow and **compose functionality from multiple small, focused classes**.

Breaking Code into Modules

Starting a new project often begins with all your code in a single script. This is fine for quick and small tasks, but as your project grows, keeping everything in one file becomes messy and overwhelming. To keep your code organized and easier to understand, it's a good idea to move functionality into separate files, also referred to as (sub)modules. Separating code into modules makes your project easier to navigate and can lay the foundation for your own **library of reusable functions and classes**, useful across multiple projects.

A typical first step is splitting the main logic of your analysis (`main.py`) from general-purpose helper functions (`utils.py`). Over time, as `utils.py` expands, you'll notice clusters of related functionality that can be moved into their own files, such as `data_utils.py`, `models.py`, or `results.py`. To create cohesive modules, you should **group code that tends to change together**, which increases maintainability as you don't need to switch between files when implementing a new feature. Modules based on domains or use cases, instead of technical layers, are therefore preferred, as the changes required to implement a new feature are generally limited to a single domain [23].

This approach naturally leads to a clean directory structure, which might look like this for a larger Python project:[3]

[3]The `__init__.py` file is needed to turn a directory into a package from which other scripts can import functionality. Usually, the file is completely empty.

```
src/
   main.py
   my_library/
       __init__.py
       data_utils.py
       models/
           __init__.py
           baseline_a.py
           baseline_b.py
           interface.py
           my_model.py
       results.py
```

In `main.py`, you can import the relevant classes and functions from these modules to keep the main script clean and focused:

```
from my_library.models.my_model import MyModel
from my_library.data_utils import load_data

if __name__ == '__main__':
    # steps that will be executed when running `python main.py`
    model = MyModel()
```

> Keep helper functions separate
>
> Always **separate reusable helper functions from the main executable code**. This also means that files like `data_utils.py` should not include a *main* function, as they are not standalone scripts. Instead, these modules provide functionality that can be imported by other scripts—just like external libraries such as `numpy`.

As you tackle more projects, you may develop a set of functions that are so versatile and useful that you find yourself reusing them across multiple projects. At that point, you might consider packaging them as your own **open-source library**, allowing other researchers to install and use them as well.

Summary: From Working to Good Code

With these best practices in mind, revisit the steps you outlined when working backward from your desired output (in our case a plot) to the necessary inputs (data):

1. **Group related steps into reusable functions.** Functions like `load_data`, `fit_model`, `predict_with_model`, and `compute_r2` help structure your code and prevent redundancy (DRY principle).
2. **Identify explicit and implicit inputs and outputs:**
    - **Inputs**: Passed as function arguments or read from external sources (files, databases, APIs—but avoid global variables!).

- **Outputs**: Return values or data written to an external source (and other side effects, like modifying input arguments).

3. **Extract pure functions**, which use only explicit inputs (arguments) and outputs (return values), from functions that rely on external resources or have other side effects. For example, if `load_data` includes both file I/O and preprocessing, separate out `preprocess` as a pure function to improve testability. Additionally, consider opportunities for **dependency injection**.
4. **Encapsulate related variables and functions into classes**:
    - Look for **multiple variables describing the same object** (e.g., parameters describing a `Model` instance).
    - Identify **functions that need access to private attributes** or should update attributes in-place (e.g., `fit` and `predict` should be methods of `Model`).
5. **Generalize where possible**:
    - Should **hardcoded values** be passed as function arguments?
    - Could multiple classes implement a **unified interface**? For example, different model classes should all implement the same `fit` and `predict` methods so they can be used with the same analysis code. They might also share some functionality through **mixins**.
6. **Organize your code into modules** (i.e., separate files and folders) when it grows too large:
    - Keep closely related functions together (e.g., `load_data` and `preprocess`, which can be expected to change together).
    - Place logically grouped files into separate directories (e.g., `models/` for different model implementations).

By following these steps, you'll create code that is not only functional but also maintainable and extensible. However, **avoid overengineering** by trying to predict every possible future requirement. Instead, keep your code **as simple as possible** and refactor it **as actual needs evolve**.

3.3. Draw Your Code

To summarize the overall design, we can create sketches that serve as both a high-level overview and an implementation guide. While formal modeling languages like UML exist for this purpose, don't feel pressured to use them—these diagrams are for you and your collaborators, so **prioritize clarity over formality**. Unless they're part of official documentation that must meet specific standards, a quick whiteboard sketch is often a better use of your time.

We distinguish between two types of diagrams:

1. **Structural diagrams** – These show the organization of your code: which functions and classes are in which modules, and how they depend on one another.
2. **Behavioral diagrams** – These describe how your program runs: how inputs flow through the system and are transformed into outputs.

Structure: Your Personal Code Library

Our first sketch provides an overview of the **reusable functions and classes** we assembled in the previous section (Figure 3.12). This collection forms your **personal code library**, which you can reuse not just for this project but for future ones as well.

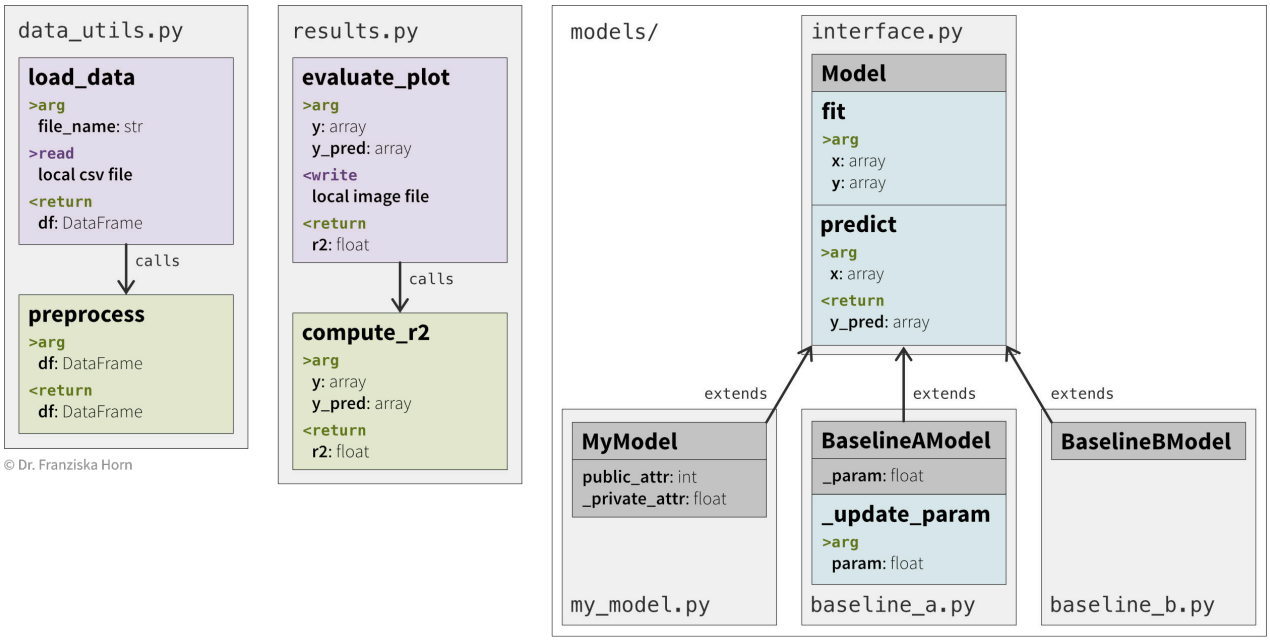


Figure 3.12.: The different functions and classes in our personal library, a collection of reusable units that can serve us in multiple projects. They are organized in different files and folders to keep related code close together. Green boxes represent pure functions, purple boxes impure functions with side effects (like reading or writing to external files), and blue boxes class methods. For classes (like `MyModel`) that extend another class (`Model`), only the additional attributes and methods are listed for that class. The arrows indicate dependencies.

Behavior: Mapping Out Your Script

Our second sketch outlines the **script you'll execute** to create the results you want, which builds upon these components (Figure 3.13). When designing the flow of your script, consider:

- **How the script is triggered** – Does it take command-line arguments? What inputs does it require?
- **What the final output should be** – A results plot? A summary table? Something else?
- **Which intermediate results should be saved** – If your script crashes, you don't want to start over. Since variables in memory disappear when the script terminates, consider saving key outputs to disk at logical checkpoints.
- **Which functions and classes from your library are used at each step** – Also, note any external resources (e.g., files, databases, or APIs) they interact with.

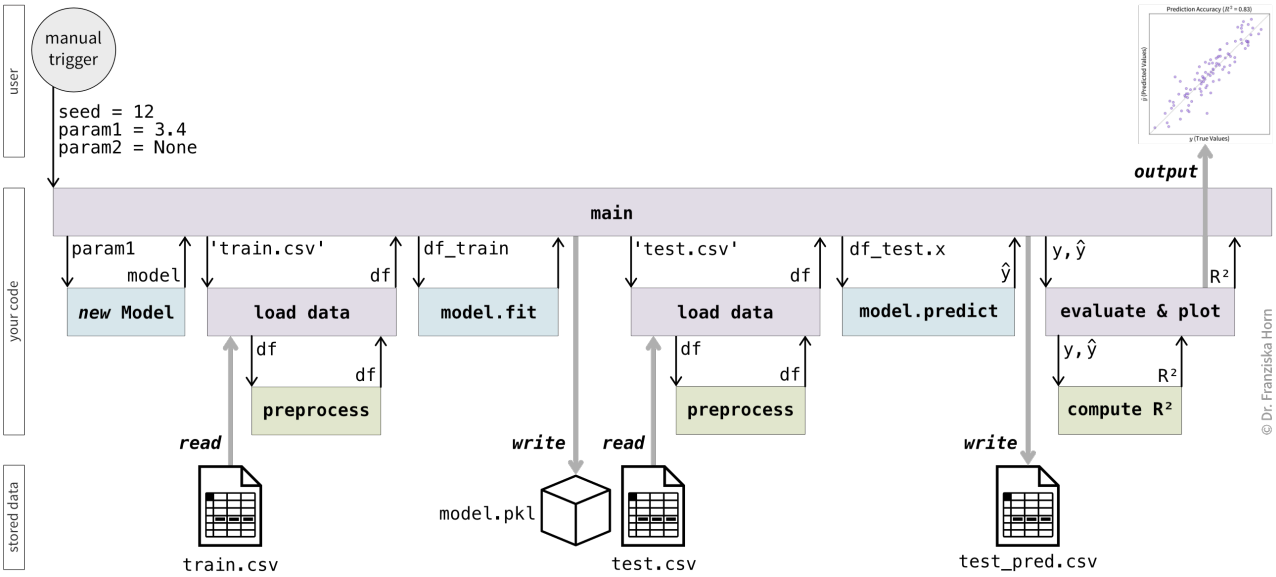


Figure 3.13.: The full flow of your script, including external resources and reusable functions. If you want, you can omit the lower level function calls as these are already covered in the library diagram. The code to create a new model should create models of different types depending on the configuration parameters passed when calling the script so you can use the same script for multiple experiments.

Before you continue

At this point, you should have a clear understanding of:

- The inputs and transformations required to produce your desired outputs—what constitutes *working code*.
- How to structure this code into *good code* by creating cohesive, decoupled, and reusable units that use simple interfaces to abstract away complicated implementation details.

Part II.

Writing Code

Now that you've gained clarity on your concept, it's finally time to write code (Figure 3.14).

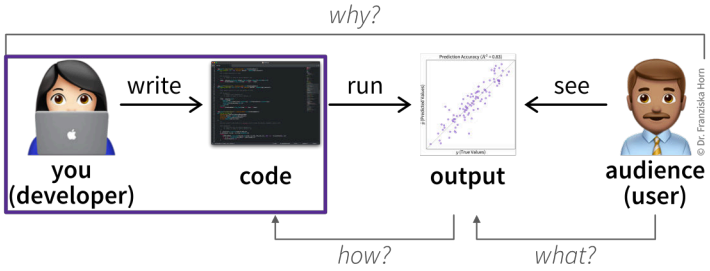


Figure 3.14.: It's finally time to implement your solution—while following some best practices.

This part starts with an introduction to some tools that will help you develop software more efficiently (Chapter 4). Next, you'll learn about the best practices you should follow during the actual implementation (Chapter 5). We'll close with an outlook on what it takes to go from research to production-grade software (Chapter 6).

4. Tools

Let's start with a quick tour of some tools that can make your software engineering journey smoother.

4.1. Programming Languages

Different programming languages suit different needs. Here's a brief overview of some popular ones used in science and engineering:

- **R**: Commonly used for statistics, with rich functionality to create data visualizations, fit statistical models (like different types of regression), and conduct advanced statistical tests (like ANOVA). The poplar Shiny framework also makes it possible to create interactive dashboards that run as web applications.
- **MATLAB**: Once dominant in engineering, used for simulations. But due to its high licensing costs, MATLAB is being replaced more and more by Python and Julia.
- **Julia**: Gaining traction in scientific computing for its speed and modern syntax.
- **Python**: A versatile language with strong support for data science, AI, web development, and more. Its active open source community has created many popular libraries for scientific computing (numpy, scipy), machine learning (scikit-learn, TensorFlow, PyTorch), and web development (FastAPI, streamlit).

Due to its broad applicability and popularity in industry, Python is used for the examples in this book. However, you should **choose the programming language that is most popular in your field** as this will make it easier for you to find relevant resources (e.g., tailored libraries) and collaborate with colleagues.

There are plenty of great books and other resources available to teach you programming fundamentals, which is why this book focuses on higher level concepts. Going forward we'll assume that you're familiar with the basic syntax and functionality of your programming language of choice (incl. key scientific libraries). For example, to learn Python essentials, you can work through this tutorial.

4.2. Version Control

Version control is a system used to **track and record changes to files** over time, like a time machine that lets you revert to any version of your code or examine how it evolved. Version control is essential in software development to keep track of code changes and collaborate effectively.

Why Use Version Control?

- **Track changes**: See what you've modified and when, with the ability to revert if necessary.
- **Review collaborators' changes**: When working with others, reviewing their changes before they are merged with the main version of the code (in so-called pull or merge requests) ensures quality and provides opportunities to teach each other better ways of doing things.
- **Not just for code**: Version control can be used for any kind of file. While it's less effective for binary formats like images or Microsoft Word documents where you can't create a clean "diff" between two versions, you should definitely give it a try when writing your next paper in a text-based format like LaTeX.

Git

The go-to tool for version control is **Git**. While desktop clients exist, you can also use `git` directly in the terminal as a command line tool.

If you're new to Git, this beginner's guide is a great place to start.

Essential git commands

- `git init`: Start a new repository in the current folder.
- `git status`: View changes.
- `git diff`: View differences between file versions before committing.
- `git add [file]`: Stage files for a commit.
- `git commit -m "message"`: Save staged changes.
- `git push`: Upload changes to a remote repository (e.g., on GitHub).
- `git pull`: Download changes from a remote repository.
- `git branch`: Create or list branches.
- `git checkout [branch]`: Switch branches.
- `git merge [branch]`: Combine branches.

By default, your repository's files are on the *main* branch. Creating a new branch is like stepping into an alternate universe where you can experiment without affecting the main timeline. **When making a major change** or adding a new feature, it's good practice to **create a new branch**, like *new-feature*, and implement your changes there. Once you're satisfied with the result, you can merge the changes back into the *main* branch.

This approach keeps the *main* branch stable and ensures you always have a working version of your code. If you decide against your new feature, you can simply abandon the branch and start fresh from *main*. By **creating a merge request** (MR) once your *new-feature* branch is ready, you or a collaborator can **review the changes** thoroughly before merging them into *main*.

To **publish your code** or **collaborate with others**, your repository (i.e., the folder under version control) can be **hosted on a platform** like:

- **GitHub**: Great for open-source projects and public personal repositories to show off your skills.
- **GitLab**: Supports self-hosting, making it ideal for organizational needs.

We strongly encourage you to publish any code related to your publications on one of these platforms to promote reproducibility of your results!

> Data versioning
>
> In addition to the changes made to your code, you should also keep track of how your data is generated and transformed over time (*data lineage*). While small datasets can be included in your repository (e.g., in a separate `data/` folder), there are also more tailored tools available specifically to version your data, like DVC.

4.3. Development Environment

The program you choose for writing code directly impacts your productivity. While you can technically write code using a plain text editor (like Notepad on Windows or TextEdit on macOS), **special-purpose text editors** and **integrated development environments (IDEs)** provide a tailored experience that boosts productivity.

Text Editors

Developer-focused text editors are lightweight tools with features like syntax highlighting and extensions for basic programming tasks.
Examples include:

- **Sublime Text**: Lightweight and fast, with excellent customization through lots of plugins.
- **Vim** and **Emacs**: Some of the first code editors, often used as command line tools and beloved by keyboard shortcuts enthusiasts.

Terminal

When you write code in a text editor, you need a way to execute it. This is where the **terminal** comes in. A terminal, or console, lets you interact with your computer through the **command line**, using text-based commands. Think of it like stepping back to the 1970s—or like being one of those cool hackers you see on TV.

A terminal app already comes preinstalled on macOS, Linux, and newer Windows versions. Inside the terminal, there's a **shell**: the actual program that processes the commands you type. The most common shells on Unix systems are `bash` and `zsh`, which are quite similar. For this book, we'll assume you're using one of these.

With the shell, you can navigate your computer's file system and run programs through their command-line interface (CLI). Try it out!

Basic shell commands

Follow along by typing these commands into your terminal. In parallel, you can watch your normal file browser to see files and folders appear or disappear as you go.

- `pwd`: Print the current working directory—this shows the path to where you opened the terminal.
- `ls`: List files and directories in the current location. Use `ls -la` for more details, including hidden files (like `.gitignore`).
- `cd path/to/folder`: Change directory to the specified path. Tip: Press the tab key to autocomplete names. If the path starts with `/`, it's absolute (from the file system's root). If it starts with `~/`, it's relative to your home directory. Use `..` to move up one folder.
- `mkdir new_folder`: Create a new directory named `new_folder`.
- `touch new_file.txt`: Create an empty file named `new_file.txt`.
- `cp new_file.txt copied_file.txt`: Copy `new_file.txt` to `copied_file.txt`. Use `mv` instead of `cp` to move or rename files.
- `rm new_file.txt`: Delete `new_file.txt`. Add `-r` to delete directories. But be careful: files deleted this way bypass the trash and are gone for good, so double-check before hitting enter!

You can also run other CLI programs in the terminal, like using the `git` commands described earlier.
A Python script can be executed with `python script.py` (assuming the script is in your current directory).

Not all CLI programs mentioned in this book will be preinstalled on your machine. Linux systems already come with a command-line package manager (like `apt` on Ubuntu), which can be used to install other tools. A popular package manager for macOS is `brew`, while for Windows you can use `winget`.

Once you get comfortable with your shell, you can also create **shell scripts** (files with a `.sh` extension) to automate tasks and handle more complex workflows. These scripts can include conditionals, loops, and other programming constructs. For more information on bash scripting, check out this resource and read the first few chapters of the book *Research Software Engineering with Python* [16].

Integrated Development Environments (IDEs)

Full IDEs combine all the tools you need in one place—file browser, editor, terminal, Git support, debugger, AI assistants, and more. They are ideal for larger projects and provide support for more complex tasks, like renaming variables across multiple files when you're refactoring code.
Examples include:

- **VS Code**: Minimalist by default but highly customizable with plugins, making it suitable for everything from basic editing to full-scale development.
- **Zed**: Similar to VS Code, with a focus on performance and collaboration.

- **JetBrains IDEs** (e.g., PyCharm): IDEs tailored to the needs of specific programming languages with very advanced features. You need to purchase a license to use the full version, but for many IDEs there is also a free community edition available.
- **JupyterLab**: An extension of Jupyter notebooks (see below), popular for data science and exploratory coding.
- **RStudio**: Tailored for R programming, with excellent support for data visualization, markdown reporting, and reproducible research workflows.
- **MATLAB**: The MATLAB programming language and IDE are virtually synonymous. However, its rich feature set comes with steep licensing fees.

Jupyter Notebooks

Jupyter notebooks are a unique format that lets you **mix code, output (like plots), and explanatory text** in one document. The name *Jupyter* is derived from Julia, Python, and R, the programming languages for which the notebook format, and later the JupyterLab IDE, were created. The IDE itself runs inside your web browser.

Notebooks are great for exploratory data analysis and to create reproducible reports. However, since the notebooks themselves are composed of individual interactive cells that can be executed in any order, developing in notebooks often becomes messy quickly. I recommend that you keep the main logic and reusable functions in separate scrips or libraries and primarily use notebooks to create plots and other results. It is also good practice once you're finished to restart the kernel and run your notebook again from top to bottom to make sure everything still works and you're not relying on variables that were defined in now-deleted cells, for example.

> Notebooks as text files
>
> Jupyter notebooks, stored as files ending in `.ipynb`, are internally represented as JSON documents. If you have your notebooks under version control (which you should), you'll notice that the diffs between versions look quite bloated. But do not despair! Tools like Jupytext can convert notebooks into plain text without loss of functionality.

> Parameterize notebooks
>
> If you want to execute the same notebook with multiple different parameter settings (e.g., create the same plots for different model configurations), have a look at papermill.

In addition to the original JupyterLab IDE and notebooks that you install on your computer, there are also free cloud-based options available, such as Google Colab, which even gives you free compute time on GPUs.

4.4. Reproducible Setups

"It works on my machine" isn't good enough for science. Reproducibility means your results can be replicated by others (and by you a few months later when the reviewers of your paper request changes to your experiments). The first step to achieve this is to **manage your dependencies** (i.e., external

libraries used by your code) to ensure the environment in which your code is executed is identical for everyone that runs your code, every time. This can be done using **virtual environments**, or, if you want to go even further, **containers** like Docker, which will be discussed in Chapter 6.

Virtual environments in Python with `uv`

Virtual environments isolate your project's dependencies, thereby ensuring consistency. For Python, a common tool to do this is uv. It tracks the libraries and their versions in a `pyproject.toml` file like this:

```
[project]
name = "example-project"
version = "0.1.0"
description = "A sample Python project"
authors = [{name="Your Name", email="youremail@example.com"}"]
requires-python = ">=3.10"
dependencies = [
    "matplotlib >=3.7.2",
    "numpy >=1.22.3,<2",
]
```

Basic commands:

- `uv init example-project`: Create a new project (folder incl. `pyproject.toml` file).
- `uv add [package]`: Add a dependency (can also be done directly in the file).
- `uv sync`: Install all dependencies.
- `uv run python script.py`: Run a Python script inside the virtual environment.

Handling Randomness

Your program will often depend on randomly sampled values, for example, when defining the initial conditions for a simulation or initializing a model before it is fitted to data (like a neural network). To ensure that your experiments can be reproduced, it is important that you always **set a random seed** at the beginning of your program so the random number generator starts from a consistent state.

Setting random seeds in Python

At the beginning of your script, set a random seed (depending on the library that you're using this can vary):

```
import random
import numpy as np

random.seed(42)
np.random.seed(42)
```

To get a better idea of how much your results depend on the random initialization and therefore how robust they are, it is advisable to always **run your code with multiple random seeds and compare the results** (e.g., compute the mean and standard deviation of the outcomes of different runs like in Figure 2.12).

> **Random state at startup**
>
> Depending on the programming language that you're using, if you run a script without executing any other code before, the random number generator may or may not always start in the same state. This means, if you don't set a random seed and, for example, run your script ten times from scratch, you may always receive the same result even though the results would differ if the code was run under different circumstances. To avoid surprises, you should always explicitly set the random seed to have more control over the results.

> **Hardware differences**
>
> If your code is run on very different hardware, e.g., a CPU vs. a GPU (graphics card, used to train neural network models, for example), despite setting a random seed, your results might still differ slightly. This is due to how the different architectures internally represent float values, i.e., with what precision the numbers are stored in memory.

4.5. Clean and Consistent Code

Especially when working together with others, it can be helpful to **follow to a style guide to produce clean and consistent code**. Google published their style guides for multiple programming languages, which is a great resource and adhering to these rules will also help you to avoid common sources of bugs.

Formatters & Linters

Since programmers are often rather lazy, they developed tools that automatically fix your code to implement these rules where possible:

- **Formatters** rewrite code to follow a consistent style (e.g., add whitespace after commas).
- **Linters** analyze code for errors, inefficiencies, and deviations from best practices.

> **Formatter & linter in Python: `ruff`**
>
> `ruff` is a (super fast) formatter and linter for Python, written in Rust. You can install it with `uv` and configure it in the same `pyproject.toml` file that we also used to manage the dependencies of our project. Then run it over your code like this:

```
ruff check        # see which errors the linter finds
ruff check --fix  # automatically fix errors where possible
ruff format       # automatically format the code
```

You'll probably want to add exceptions for some of the errors that the linter checks for in your `pyproject.toml` file as `ruff` is quite strict.

It is important to have the configuration for your formatter and linter under version control as well, so that all collaborators use the same settings and you avoid unnecessary changes (and bloated diffs in merge requests) when different people format the code.

Pre-commit Hooks

In the heat of the moment, you might forget to run the formatter and linter over your code before committing your changes. To **avoid accidentally checking messy code into your repository**, you can configure so-called "pre-commit hooks". Pre-commit hooks catch issues automatically by enforcing coding standards before committing or pushing code with git.

Setting up pre-commit hooks

First, you need to install pre-commit hooks, e.g., through Python's package manger `pip`:

```
pip install pre-commit
```

Then configure it in a file named `.pre-commit-config.yaml` (here done for `ruff`):

```
repos:
- repo: https://github.com/pre-commit/pre-commit-hooks
  rev: v2.3.0
  hooks:
    - id: check-yaml
    - id: end-of-file-fixer
    - id: trailing-whitespace
- repo: https://github.com/astral-sh/ruff-pre-commit
  # Ruff version
  rev: v0.8.3
  hooks:
    # Run the linter
    - id: ruff
      args: [ --fix ]
    # Run the formatter
    - id: ruff-format
```

Then install the git hook scripts from the config file:

```
pre-commit install
```

Now the configured hooks will be run on all changed files when you try to commit them and you can only proceed if all checks pass.

To catch any style inconsistencies after the code was pushed to your remote repository (e.g., in case one of your collaborators has not installed the pre-commit hooks), you can also add these checks to your CI/CD pipeline (see Chapter 6).

4.6. Putting It All Together

When you set up all these tools, your repository should look something like this (see here for more details; setup for programming languages other than Python will differ slightly):

```
project-name/
   .gitignore              # Exclude unnecessary files from version control
   README.md               # Describe the project purpose and usage
   pre-commit-config.yaml  # Pre-commit hook setup
   pyproject.toml          # Python dependencies and configs
   data/                   # Store (small) datasets
   notebooks/              # For exploratory analysis
   src/                    # Core source code
   tests/                  # Unit tests
```

A clean project structure makes it easier to maintain your code.

> **Before you continue**
>
> At this point, you should have a clear understanding of:
>
> - How to set up your development environment to code efficiently.
> - How to host your version-controlled repository on a platform like GitHub or GitLab, complete with pre-commit hooks to ensure well-formatted code.
> - The fundamental syntax of your programming language of choice (incl. key scientific libraries) to get started.

If you want to gain a deeper understanding of many of the tools mentioned here, along with additional techniques, have a look at the book *Research Software Engineering with Python* [16].

5. Implementation

Armed with the right tools and a solid concept, it's time to start coding. In this chapter we'll discuss some practical tips and best practices to ensure your ideas do not only look good on paper.

5.1. Make a Plan

Seeing your software design mapped out in full can feel overwhelming—there's so much to do! That's why it's important to **break the implementation process into small, manageable tasks.** Rather than getting stuck in the details, **start by implementing the full end-to-end flow**. Once that's in place, you can refine individual models, tweak data processing steps, or make plots prettier.

> Minimum Viable Results
>
> In product development, there's a concept called the **Minimum Viable Product (MVP)**. This refers to the simplest version of a product that still provides value to users.
> The MVP serves as a prototype to gather feedback on whether the product meets user needs and to identify which features are truly essential. By iterating quickly and testing assumptions, teams can increase the odds of creating a successful product that people will actually pay for. This approach also has motivational benefits. Seeing something functional—even if basic—early on makes it easier to stay engaged. It's far better than toiling for months without tangible results.
> You should apply the same mindset to your research software development by **starting with a script that generates "Minimum Viable Results."** This means creating a program that produces outputs resembling your final results, like plots or tables, but using placeholder data instead of actual values. For instance:
>
> - If your goal is to build a prediction model, start with one that simply predicts the mean of the observed data.
> - If you're developing a simulation, begin with random outputs, such as a random walk.
>
> By starting with Minimum Viable Results, you can **test your code end-to-end** early on, see tangible progress, and **iteratively improve** from there.
> This approach also serves as a **"stupid baseline"**—a simple, easy-to-beat reference point for your final method. It's a sanity check: if your sophisticated model can't outperform a baseline that always predicts the mean, something's off.

To organize implementation steps and track progress, consider using a **Kanban board**, a tool commonly used in project management. Software like Trello, Notion, or Linear can help you create Kanban boards, where you can describe tasks in more detail (e.g., adding sketches of the plots you want to generate) compared to a simple to-do list.

5.2. From Concept to Code

Using your software design from Chapter 3, you can now **translate your sketch into a code skeleton**. Start by outlining the functions, place calls to them where needed, and add comments for any steps you'll figure out later. For example, the design from Figure 3.13 could result in the following draft:

```
import numpy as np
import pandas as pd

# in models/my_model.py
class MyModel:
    def __init__(self, param1):
        self.param1 = param1

    def fit(self, x, y):
        pass

    def predict(self, x):
        y = ...
        return y

# in data_utils.py
def preprocess(df):
    df = ...
    return df

def load_data(file_name):
    df = pd.read_csv(file_name)
    df = preprocess(df)
    return df

# in results.py
def compute_r2(y, y_pred):
    r2 = ...
    return r2

def evaluate_plot(y, y_pred):
    r2 = compute_r2(y, y_pred)
    # ... create and save plot ...
    return r2

# in main.py
def main(model):
    df_train = load_data("train.csv")
    model.fit(df_train.x, df_train.y)
    df_test = load_data("test.csv")
```

```
    y_pred = model.predict(df_test.x)
    r2 = evaluate_plot(df_test.y, y_pred)

if __name__ == '__main__':
    # script is called as `python main.py seed param1 [param2]`
    seed, param1, param2 = ...
    np.random.seed(seed)
    # TODO: check which type of model should be created (mine or baseline)
    model = MyModel(param1)
    main(model)
```

Order of functions

Your code files likely include multiple functions, so you'll need to decide their order from top to bottom. Since scripts typically start with imports (e.g., of libraries like `numpy`) and end with a `main` function, personally I prefer to put more general functions (i.e., the ones that are at the lower levels of abstraction in your call hierarchy and that only rely on external dependencies) towards the top of the file. This ensures that, as you read the script from top to bottom, **each function depends only on what was defined before it**. Maintaining this order avoids circular dependencies and encourages you to write reusable, modular functions that serve as building blocks for the code that follows.

Fill in the Blanks

Once your skeleton stands, you "only" need to **fill in the details**, which is a lot less intimidating than facing a blank page. Plus, since you started with a thoughtful design, your final program is more likely to be well-structured and easy to understand. Compare this to writing code on the fly, where decisions about functions are often made haphazardly—you'll appreciate the difference.

Using AI code generators

AI assistants like ChatGPT, Claude, or GitHub Copilot can be helpful tools when writing code, especially at the level of individual functions. However, remember that these tools only reproduce patterns from their training data, which includes both good and bad code. As a result, the code they generate may not always be optimal. For instance, they might use inefficient for-loops instead of more elegant matrix operations. Similarly, support for less popular programming languages may be subpar.
To get better results, consider crafting prompts like: *"You are a senior Python developer with 10 years of experience writing efficient, edge-case-aware code. Write a function …"*

Think it through!

It's essential that you always **understand what your code is doing** (especially if it was written with AI assistance).
Go through your code line by line and consider **what is true at each step**. This will help

you spot issues such as conditional statements that are always true or always false and therefore unnecessary and potentially confusing. For example:

```
min_value = 5

for i in [10, 100, 1000]:
    if i > min_value:  # always True
        ...
```

Here, the condition `i > min_value` is redundant—it will always evaluate to `True`. It might have made sense in an earlier version of the code when `i` could be less than `min_value`, but now it only adds confusion, making a reader think they're missing something.
While this case is obvious, other instances can be much harder to detect. So to improve your own understanding of the code—and to help others who read it—you should always know what each line does and remove any logic that no longer serves a purpose.

Keep It Compact

When writing code, aim to achieve your goals while using **as little screen space as possible**—this applies to both the number of lines and their length.

Tips to create compact, reusable code

- **Avoid duplication**: Instead of copying and pasting code in multiple places, consolidate it into a reusable function to save lines (DRY principle).
- **Prefer 'deep' functions**: Avoid extracting very short code fragments (1-2 lines) into a separate function, especially if this function would require many arguments. Such shallow functions with wide interfaces increase complexity without meaningfully reducing line count. Instead, strive for *deep functions* (spanning multiple lines) with *narrow interfaces* (e.g., only 1-3 input arguments, i.e., **fewer arguments than the function has lines of code**), which tend to be more general and reusable [26].
- **Address nesting**: If your code becomes overly nested, this can be a sign that parts of the code should be moved into a separate function. This simplifies logic and shortens lines.
- **Use guard clauses**: Deeply nested `if`-statements can make code harder to read. Instead, use guard clauses [2] to handle preconditions (e.g., checking for wrong user input) early, leaving the "happy path" clear and concise. For example:

```
if condition:
    if not other_condition:
        # do something
        return result
else:
    return None
```

Can be refactored into:

```
if not condition:
    return None
if other_condition:
    return None
# do something
return result
```

This approach reduces nesting and improves readability.

5.3. Documentation & Comments

While you write it, everything seems obvious. However, when revisiting your code a few months later (e.g., to try a different experiment), you're often left wondering what the heck you were doing. This is especially true when some external constraint (like a library quirk) forced you to create a workaround instead of opting for the straightforward solution. When returning to such code, you might be tempted to replace the awkward implementation with something more elegant, only to rediscover why you chose that approach in the first place. This is where comments can save you some trouble. And they are even more important when collaborating with others who need to understand your code.

We distinguish between documentation and comments: **Documentation** provides the general description of ***when and how to use your code***, such as function docstrings explaining what the function computes, its input arguments, and return values. This is particularly important for open source libraries where you can't personally explain the code's purpose and usage to others. **Comments** help developers understand ***why your code was written in a certain way***, like explaining that unintuitive workaround. Additionally, for scientific code, you may also need to document the origin of certain values or equations by referencing the corresponding paper in the comments.

⚠ Code should be self-documenting

Ideally, your code should be written so clearly that it's self-explanatory. Comments shouldn't explain *what* the code does, only *why* it does that (when not obvious). **Comments and documentation, like code, need to be maintained**—if you modify code, you need to update the corresponding comments or they become misleading and harmful rather than helpful. Using comments sparingly minimizes the risk of confusing, outdated comments.
Informative variable and function names are essential for self-explanatory code. When you're tempted to write a comment that summarizes what the following block of code does (e.g., `# preprocess data`), consider **moving these lines into a separate function with an informative name**, especially if they contain significant, reusable logic.

Naming Is Hard

> *There are only two hard things in Computer Science: cache invalidation and naming things.*
> – Phil Karlton[1]

Finding informative names for variables, functions, and classes can be challenging, but good names are crucial to make the code easier to understand for you and your collaborators.

Tips for effective naming

- Names should **reveal intent**. Longer names (consisting of multiple words in `snake_case` or `camelCase`, depending on the conventions of your chosen programming language) are usually better. However, **stick to domain conventions**—if everyone understands `X` and `y` as feature matrix and target vector, use these despite common advice denouncing single letter names.
- **Be consistent**: similar names should indicate similar things. This makes it easier to **recognize patterns** and extrapolate what you've learned about one implementation (e.g., `BaselineAModel`) to others (e.g., `BaselineBModel`), which reduces the **mental effort required to understand the code** [22]. On the other hand, if you name two things similarly even though they behave differently, this increases the cognitive load as you need to explicitly memorize this exception.
- **Avoid reserved keywords** (i.e., words your code editor colors differently, like Python's `input` function).
- Use **verbs for functions, nouns for classes**.
- Use **affirmative phrases for booleans** (e.g., `is_visible` instead of `is_invisible`).
- Use **plurals for collections** (e.g., `cats` instead of `list_of_cats`).
- **Avoid encoding types in names** (e.g., `sample_array`), since if you decide to change the data type later, you either need to rename the variable everywhere or the name is now misleading.
- **Use AI to generate naming suggestions** by describing the concept you're trying to represent and asking for a suitable name.

5.4. Testing

We all want our code to be correct. During development, we often verify this manually by running the code with example inputs to check if the output matches our expectations. While this approach helps to ensure correctness initially, it becomes cumbersome to recreate these test cases later when the code needs changes. The simple solution? **Package your manual tests into a reusable test suite** that you can run anytime to check your code for errors.

Tests typically use `assert` statements to confirm that the actual output matches the expected output. For example:

[1] https://martinfowler.com/bliki/TwoHardThings.html

```python
def add(x, y):
    return x + y

def test_add():
    # verify correctness with examples, including edge cases
    # syntax: assert (expression that should evaluate to True), "error message"
    assert add(2, 2) == 4, "2 + 2 should equal 4"
    assert add(5, -6) == -1, "5 - 6 should equal -1"
    assert add(-2, 10.6) == 8.6, "-2 + 10.6 should equal 8.6"
    assert add(0, 0) == 0, "0 + 0 should equal 0"
```

Testing in Python with `pytest`

Consider using the `pytest` framework for your Python tests. Organize all your test scripts in a dedicated `tests/` folder to keep them separate from the main source code.

As discussed in Chapter 3, **pure functions**—those without side effects like reading or writing external files—are especially **easy to test** because you can directly supply the necessary inputs. Placing your main logic into pure functions therefore simplifies testing the critical parts of your code. For impure functions, such as those interacting with databases or APIs, you can use techniques like **mocking** to simulate external resources, ideally combined with **dependency injection**.

When designing your tests, focus on **edge cases**—unusual or extreme scenarios like values outside the normal range or invalid inputs (e.g., dividing by zero or passing an empty list). The more thorough your tests, the more confident you can be in your code. Each time you make significant changes, run all your tests to ensure the code still behaves as expected.

Some developers even adopt **Test-Driven Development (TDD)**, where they write tests before the actual code. The process begins with writing tests that fail (or don't even compile), then creating the code to make them pass. TDD can be highly motivating as it provides clear goals, but it requires discipline and may not always be practical in the early stages of development when function definitions are still evolving.

Testing at different levels

Ideally, you'll test your software at all levels:

- **Unit tests:** Test individual units (e.g., single functions) to verify basic logic. These should make up the bulk of your test code.
- **Integration/system tests:** Check that different parts of the system work together as expected. These often require more complex setups, like running multiple services at the same time, to test the flow end-to-end.
- **Manual testing:** Identify unexpected behavior or overlooked edge cases. **Whenever a bug is found, create an automated test to reproduce it and prevent regression.**
- **User testing:** Evaluate the user interface (UI) with real users to ensure clarity and usability. UX designers often perform these tests using design mockups before coding begins.

5.5. Debugging

When your code doesn't work as intended, you'll need to debug—**systematically identify and fix the problem**. Debugging becomes easier if your code is organized into small, testable functions covered by unit tests. These tests often help narrow down the source of the issue. If your existing tests didn't catch the bug, **first write a new, failing test to reproduce it** before fixing it. This confirms your fix works and prevents a regression—meaning the bug won't sneak back into the code later.

To isolate the exact line causing the error:

- Use `print` statements to log variable values at key points and understand the program's flow.
- Add `assert` statements to verify intermediate results.
- Use a **debugger**, often integrated into your IDE, to set breakpoints where execution will pause, allowing you to step through the program manually and inspect variables.

Debugging is an essential skill, not only to identify the root cause of bugs, but also to improve your understanding of the code and its behavior.

5.6. Optimizing Performance

> *Make it run, make it right, make it fast.*
> – Kent Beck (or rather his dad, Douglas Kent Beck[2])

Now that your code works and produces the right results (as you've dutifully confirmed with thorough testing), it's time to think about performance.

> **!** Readability over performance
>
> **Always prioritize writing code that's easy to understand.** Performance optimizations should not come at the cost of readability. More time is spent by humans reading and maintaining code than machines executing it.

Find and Fix the Bottlenecks

Instead of randomly trying to speed up everything, focus on the parts of your code that are actually slow. A simple way to identify bottlenecks is to insert log statements with timestamps at key points in your code to measure the time elapsed between steps. And if you manually interrupt a (Python) script during a long run and it always stops in the same place, that's also often an indication that this step could be the issue. For a more systematic approach, use a **profiler**. Profilers analyze your code and show you how much time each part takes, helping you decide where to focus your efforts.

Accessing files on disk or fetching data over the network is one of the slowest operations in most programs. Whenever possible, **cache the results** by storing the loaded data in memory to avoid

[2] https://x.com/KentBeck/status/704385198301904896

repeated access to external resources. Just be mindful of how frequently the external data changes and invalidate the cache when the information becomes outdated.

> **Run it in the cloud**
>
> Working with large datasets may trigger `Out of Memory` errors as your computer runs out of RAM. While optimizing your code can help, sometimes the quickest solution is to run it on a larger machine in the cloud. Platforms like AWS, Google Cloud, Azure, or your institution's own compute cluster make this cost-effective and accessible. That said, always look for simple performance improvements first!

Think About Big O

Some computations have unavoidable limits. For example, finding the maximum value in an unsorted list requires checking every item—there is no way around this. The "Big O" notation is used to describe these limits, helping you understand how your code scales as data grows (both in terms of execution time and required memory).

- **Constant time** ($\mathcal{O}(1)$): Independent of dataset size (e.g., looking up a key in a dictionary).
- **Linear time** ($\mathcal{O}(n)$): Grows proportionally to data size (e.g., finding the maximum in a list).
- **Problematic growth** (e.g., $\mathcal{O}(n^3)$ or $\mathcal{O}(2^n)$): Polynomial or exponential scaling can make algorithms impractical for large datasets.

When developing a novel algorithm, you should **examine its scaling behavior both theoretically (e.g., using proofs) and empirically (e.g., timing it on datasets of different sizes)**. Designing a more efficient algorithm is a major achievement in computational research!

Divide & Conquer

If your code is too slow or your dataset too large, try **splitting the work into smaller, independent chunks and combining the results**. Such a "divide and conquer" approach is used in many algorithms, like the merge sort algorithm, and in big data frameworks like MapReduce.

Example: MapReduce

MapReduce [6] was one of the first frameworks developed to work with 'big data' that does not fit on a single computer anymore. The data is split into chunks and distributed across multiple machines, where each chunk is processed in parallel (*map* step), and then the results are combined into the final output (*reduce* step).

For instance, if you want to train a machine learning model on a very large dataset, you could train separate models on subsets of the data and then aggregate their predictions (e.g., by averaging them), thereby creating an ensemble model.

Replace for-loops with map/filter/reduce

Sequential `for` loops can often be replaced with `map`, `filter`, and `reduce` operations for better readability and potential parallelism:

- `map`: Transform each element in a sequence.
- `filter`: Keep elements that meet a condition.
- `reduce`: Aggregate elements recursively (e.g., summing values).

For example:

```
from functools import reduce

### Simplify this loop:
result_sum = 0
result_max = -float('inf')
for i in range(10000):
    new_i = i**0.5
    # the modulo operator x % y gives the remainder when diving x by y
    # i.e., we're checking for even numbers, where the rest is == 0
    if (round(new_i) % 2) == 0:
        result_sum += new_i
        result_max = max(result_max, new_i)

### Using map/filter/reduce:
# map(function to apply, list of elements)
new_i_all = map(lambda x: x**0.5, range(10000))
# filter(function that returns true or false, list of elements)
new_i_filtered = filter(lambda x: (round(x) % 2) == 0, new_i_all)
# reduce(function to combine current result with next element, list of elements, initial val
result_sum = reduce(lambda acc, x: acc + x, new_i_filtered, 0)
result_max = reduce(lambda acc, x: max(acc, x), new_i_filtered, -float('inf'))
# (of course, for these simple cases you could just use sum() and max() on the list directly
```

In Python, list comprehensions also offer concise alternatives:

```
new_i_filtered = [i**0.5 for i in range(10000) if (round(i**0.5) % 2) == 0]
```

Exploit Parallelism

Many scientific computations are "**embarrassingly parallelizable**," meaning tasks can run independently. For example, running simulations with different model configurations, initial conditions, or random seeds. Each of these experiments can be submitted as a separate job and run in parallel on a compute cluster. By identifying parts of your code that can be parallelized, you can save time and make full use of available resources.

5.7. Refactoring

Refactoring is the process of **modifying existing code without altering its external behavior** [10]. In other words, it preserves the "contract" (interface) between your code and its users while improving its internal structure.
Common refactoring tasks include:

- **Renaming**: Giving (internally used) variables, functions, or classes more meaningful and descriptive names.
- **Extracting functions**: Breaking large functions into smaller, more focused ones (with a **single responsibility**).
- **Eliminating duplication**: Consolidating repeated code into reusable functions (**DRY** principle).
- **Simplifying logic**: Reducing deeply nested code structures or introducing guard clauses for clarity.
- **Reorganizing code**: Grouping related functions or classes into appropriate files or modules.

Why Refactor?

Refactoring is typically done for two main reasons:

1. **Addressing technical debt**:
   When code is written quickly—often to meet deadlines—it may include shortcuts that make future changes harder. This accumulation of compromises is called "technical debt." Refactoring cleans up this debt, improving code quality and making the code easier to understand.
   - Example: Revisiting old code can be like tidying up a messy campsite. Just as a good scout leaves the campground cleaner than they found it, a responsible developer leaves the codebase better for the next person (or themselves in the future).
2. **Making change easier**:
   Sometimes, implementing a new feature in your existing code feels like forcing a square peg into a round hole. Instead of struggling with awkward workarounds, you should first refactor your code to align with the new requirements. The goal of software design isn't to predict every possible future change (which is impossible) but to adapt gracefully when those changes arise. This promotes an evolutionary architecture, where you solve problems once you understand them better [8].
   - Before adding a new feature, clean up your code so that the change feels natural and seamless. This not only simplifies the task at hand but also results in more general, reusable functions and classes.

Tips when refactoring

- **Test as you refactor**: Always run tests before and after refactoring to ensure no functionality is accidentally broken. Writing or expanding automated tests is often part of the process to safeguard against regressions.
- **Leverage IDE support**: Modern IDEs like PyCharm or Visual Studio Code provide tools

for automated refactoring, such as renaming, extracting functions, or moving files. These can save time and reduce errors.
- **Avoid over-refactoring**: While cleaning up code is valuable, avoid making unnecessary changes that don't improve functionality or clarity. Over-refactoring wastes time and can confuse collaborators.

Refactorings to Simplify Changes

For each desired change, make the change easy (warning: this may be hard), then make the easy change.
– Kent Beck[3]

- **Replace magic numbers with constants**: Magic numbers—values with unclear meaning—can make code harder to understand and maintain. By replacing them with constants, you create a single source of truth that's easy to modify.

```
# before:
if status == 404:
    ...

# after:
ERROR_NOT_FOUND = 404
if status == ERROR_NOT_FOUND:
    ...
```

You can also use enumerations to specify a set of related constants. Enums can be especially helpful to **make a function's interface explicit**. For example, by specifying that an input argument should be an `HTTPStatus`, users of the function know that they can't just pass any arbitrary integer:

```
from enum import Enum

class HTTPStatus(Enum):
    OK = 200
    CREATED = 201
    FORBIDDEN = 403
    NOT_FOUND = 404
    INTERNAL_SERVER_ERROR = 500
    SERVICE_UNAVAILABLE = 503

def get_status_message(status_code: HTTPStatus):
    """Returns the HTTP status message for a given HTTPStatus enum value."""
    return f"{status_code.name.replace('_', ' ').title()} ({status_code.value})"

if __name__ == "__main__":
```

[3] https://x.com/KentBeck/status/250733358307500032

```
    print(get_status_message(HTTPStatus.OK))  # Output: Ok (200)
    print(get_status_message(HTTPStatus.NOT_FOUND))  # Output: Not Found (404)
```

- **Don't Repeat Yourself (DRY)**: Copying and pasting code may seem like a quick fix, but it leads to problems later. If the logic changes, you'll need to update it everywhere it's duplicated, which is error-prone. Instead, move the logic into a reusable function or method.

```
# before:
if (model.a > 5) and (model.b == 3) and (model.c < 8):
    ...

# after:
class MyModel:
    def is_ready(self):
        return (self.a > 5) and (self.b == 3) and (self.c < 8)

if model.is_ready():
    ...
```

- **Implement wrappers**: When working with external libraries or APIs, their provided interface might not align with your needs, and adapting to it directly can lead to awkward implementations in your code. A better solution is to create a wrapper that implements **the interface you wish you had**, translating the external API's inputs and outputs into the format that best suits your implementation. This approach keeps your code clean, consistent, and easier to maintain, while confining the less-than-ideal API interactions to a single location. Plus, if the external API changes, you only need to update the wrapper instead of changing your code everywhere—which is why these wrappers are also called *anti-corruption layers*.

- **Use alternative constructors**: Similar to a wrapper, you can add a class method to create objects in a way that's different from the regular constructor. This is useful when the input data doesn't directly match what the constructor needs. For instance, imagine you have a configuration file that specifies settings for a simulation. If the names or structure of these settings don't match the constructor's parameters, you can create a `from_config` method to handle the translation and then call the constructor with the correct arguments. The advantage is that if the format of the configuration file changes in the future, you only need to update this one method, keeping the rest of your code the same.

```
class Date:
  def __init__(self, year, month, day):
      self.year = year
      self.month = month
      self.day = day

  @classmethod
  def from_str(cls, date_str):
      # parse the string and create a new instance
      year, month, day = map(int, date_str.split('-'))
      return cls(year, month, day)
```

```
# usage:
date1 = Date(2025, 01, 30)
date2 = Date.from_str("2025-01-30")
```

- **Organize for cohesion**: Keep code elements that need to change together in the same file or module. Conversely, separate unrelated parts of your code to prevent unnecessary entanglement. This way, changes are localized, which reduces cognitive load.

  In larger codebases shared by multiple teams, this is even more critical. **When changes require excessive communication and coordination between teams, it signals a need to reorganize the code.** Clear ownership and reduced dependencies help teams work independently by making sure the system is loosely coupled through agreed upon interfaces.

Large-scale Refactoring

While smaller portions of code are often tidied up as you go [2], larger refactorings that span multiple files or repositories require more upfront planning [21][22][23].

The following steps will help keep larger refactorings on track and ensure they deliver real improvements:

1. **Identify the problems this refactoring should address.**

   Don't refactor just because the code is *ugly*—refactor because it's **causing problems**. For example, the current structure may make it difficult to implement important new features or maintain the code efficiently. **List all the problems** the code creates and **rank them by severity** to ensure your refactoring tackles the most critical issues.

2. **Envision the ideal state.**

   Code can become suboptimal for many reasons. Maybe a looming deadline forced developers to take shortcuts, leading to **technical debt**. Maybe an inexperienced developer made a **design choice that no longer fits**. But most likely, the **requirements have evolved** since the code was written, making what was once a good solution no longer suitable.

   Try to break free from the existing structure and its limitations. If you were **designing the system from scratch today**, given everything you now know about **current and future requirements**, what would you build? What should the code look like once refactored?

3. **Verify that the ideal state solves the most important issues.**

   Revisit your **list of problems** and ensure that your envisioned ideal state actually **addresses them**. If necessary, **iterate on your vision** until you have a solution that tackles the most critical challenges. This gives you a better understanding of which changes to the codebase are actually necessary to achieve your goal.

4. **Make a realistic plan to get closer to your ideal state.**

   While it might be tempting to **rewrite everything from scratch**, this is rarely practical. A complete rewrite would likely take longer than expected and introduce **new, unforeseen issues**, as the existing code likely accounts for edge cases and hidden requirements you've forgotten.

Instead, **translate your ideal state into small, targeted changes** to the existing codebase that still provide **significant benefits**. Ideally, each change should:

- Be **independent** of the others, allowing for **incremental progress**.
- Deliver **some immediate value** on its own.

Create a **prioritized list of independent changes**, considering both:

- **Impact:** Which of the original problems does this change solve?
- **Effort:** How difficult would this be to implement? What dependencies or additional steps are required (e.g., database migrations, external system changes, or involvement from other teams)?

To accurately assess effort, **detail your plan**—outline **which files and functions will be affected** and identify any external dependencies. Once you've evaluated **impact vs. effort**, decide **which changes are essential**, which are nice-to-have, and which might not be worth the effort, while taking into account that some steps might depend on the successful completion of other changes.

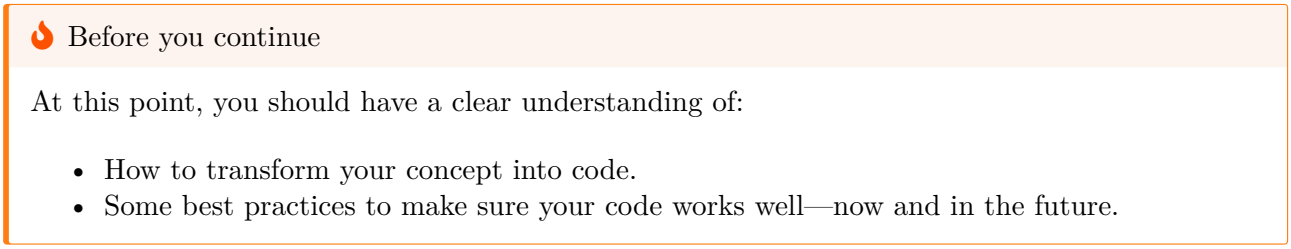
5. **Get feedback on your plan.**

   If possible, **discuss your plan with collaborators and stakeholders**—especially those affected by the changes. They might **catch overlooked dependencies** or **identify potential blockers** before you run into them during implementation.

6. **Execute incrementally and merge frequently.**

   Instead of implementing all changes at once, **work step by step, merging updates back into the codebase as quickly as possible**. This minimizes risk, ensures early **testing and validation**, and helps maintain motivation—since every small change delivers **immediate value**.

By refactoring regularly and following these practices, you'll create a cleaner, more maintainable codebase that is adaptable to future needs and enjoyable to work with.



6. From Research to Production

Your results look great, the paper is written, the conference talk is over—now you're done, right?! Well, in academia, you might be. But in industry, this is often just the beginning.

In this chapter, we explore some concepts and tools that can elevate your code to the next level: the additional components required to build a full-fledged software product that users can interact with, as well as some strategies for deploying your code and delivering it to your end users (Figure 6.1).

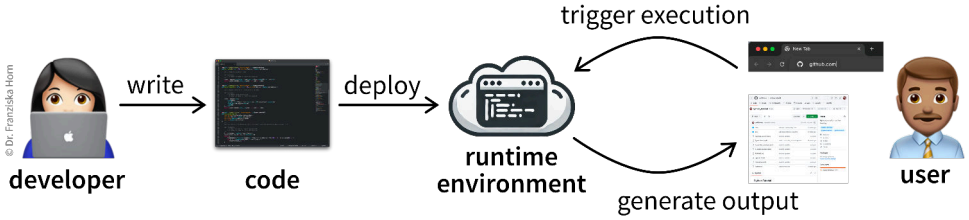


Figure 6.1.: Unlike a script that runs locally on your laptop, a production-grade software application—such as a mobile app or a cloud-based web service—requires your code to be packaged and deployed within a runtime environment. The user can then interact with the software by providing inputs (like clicking a button) and receiving outputs (like seeing a new webpage).

Who knows—maybe you'll even be inspired to turn your project into a deployable app, which could become a great showcase in your next job application.

6.1. Components of Software Products

So far, your code might consist of scripts or notebooks with analyses and a set of reusable helper functions in your personal library. The next step? Making your code accessible to others by turning it into standalone software with a graphical user interface (GUI). Additionally, we'll explore how to expand beyond static data sources like CSV or Excel files.

Graphical User Interface (GUI)

Software shines when users can interact with it easily. Instead of using a command-line interface (CLI), these days, users expect intuitive GUIs with buttons and visual elements.

We can broadly categorize software programs into:

1. **Stand-alone desktop or mobile applications**, which users download and install on their devices.

2. **Web-based applications** that run in a browser, like Google Docs. These are increasingly popular thanks to widespread internet access.

For web apps, the GUI that users interact with is also referred to as the **frontend**, while the **backend** handles behind-the-scenes tasks like data storage and processing. Even seemingly standalone desktop clients often connect to a backend server for cloud storage or to enable collaboration on shared documents. We'll explore how this works in the section on APIs.

In research, the goal is often to make results more accessible, for example, by transforming a static report into an interactive dashboard where users can explore data. To do this, we recommend you start with a web-based app.

Many books and tutorials are written on the topic of building user-friendly software applications and a lot of it is very specific to the programming language and framework you're using—please consult your favorite search engine to discover more resources on this topic.[1]

> Web apps in Python
>
> If you use Python, try the Panel or Streamlit framework to turn your analysis scripts into a web app with just a few additional lines of code. To create more advanced web apps, you can also check out the Reflex framework.

Databases

So far, we've assumed that your data is stored in spreadsheets (like CSV or Excel files) on your computer. While this works for smaller datasets and simple workflows, it becomes less practical as your data grows or is generated dynamically, such as through user interactions, and needs to be accessed and updated by multiple people at the same time. This is where databases come in, offering a more efficient and scalable way to store, retrieve, and manage data [19].

Databases come in many forms, each suited to different types of data and use cases. Two key considerations when choosing a database are [29]:

1. the kind of data you need to store, and
2. how that data will be used.

Types of Data in Databases

Different kinds of databases are ideal for different types of data (see also Section 2.2):

- **Structured data:** This resembles spreadsheet data, with rows for records and columns for attributes. Structured data is typically stored in relational (SQL) databases, where data is organized into multiple interrelated tables. Each table has a schema—a strict definition of the fields it contains and their types, such as text or numbers. If data doesn't match the schema, it's rejected.

[1] I recommend choosing a framework that follows a reactive programming style, as this often leads to simpler implementations compared to manually handling events in imperative, callback-driven frameworks.

> **Normalization in relational databases**
>
> A process called normalization reduces redundancy by splitting data into separate tables. For example, instead of storing `material_type`, `material_supplier`, and `material_quality` directly in a table of `samples`, you'd create a `materials` table with a unique ID for each material, then reference the `material_id` in the `samples` table. This avoids duplication and makes updates easier but requires more complex queries to combine tables and extract all the data needed for analysis.

- **Semi-structured data:** JSON or XML documents contain data in flexible key-value pairs and are often stored in NoSQL databases. Unlike SQL databases, these databases don't enforce a strict schema, which makes them ideal for handling complex, nested data structures or dynamically changing datasets. For example, APIs often exchange data in JSON format, which can be stored as-is to avoid breaking it into tables and reconstructing it later.
  Modern relational databases, such as Postgres, blur the line between structured and semi-structured data by supporting JSON columns alongside traditional tables.

- **Unstructured data:** Files like images, videos, and text documents are typically stored on disk. Data lakes (e.g., AWS S3) provide additional tools to manage these files, but fundamentally, this is similar to organizing files in folders on your computer. If your data is processed through a pipeline, it's often a good idea to save copies of the files at each stage (e.g., in `raw/` and `cleaned/` folders).

- **Streaming data:** High-volume, real-time data (e.g., IoT sensor logs) is best managed in specialized databases optimized for streaming, such as Apache Kafka.

Use Cases for Databases

When choosing a database, you'll also want to consider how the data will be used later:

- **Transactional processing (OLTP):** In this use case, individual records are frequently created and retrieved (e.g., financial transactions). These systems prioritize fast write speeds and maintaining an up-to-date view of the data.
- **Batch analytics (OLAP):** Data analysis is often performed in large batches to generate reports or insights, such as identifying which products users purchased last month. To avoid overloading the operational database with complex queries, data is typically copied from transactional systems into analytical systems (e.g., data warehouses) using an ETL process (Extract-Transform-Load).
- **Real-time analytics:** For applications requiring live data (e.g., interactive dashboards), databases and frameworks optimized for streaming or in-memory processing (e.g., Apache Flink) are ideal.

Scaling your database system is another critical factor. Consider how many users will access it simultaneously, how much data they'll retrieve, and how often it will be updated.

CRUD Operations

Databases support four basic operations collectively called CRUD:

- **Create:** Add new records.
- **Read:** Retrieve data.
- **Update:** Modify existing records.
- **Delete:** Remove records.

These operations are performed using **queries**, often written in SQL (Structured Query Language). For example, `SELECT * FROM table;` retrieves all records from a table. If you're new to SQL, this tutorial is a great place to start.

ORMs and Database Migrations

Writing raw database queries can be tedious, especially when working with complex schemas. **Object-Relational Mappers** (ORMs) simplify this by mapping database tables to objects in your code. With ORMs, you can interact with your database using familiar programming constructs and even define the schema directly in your code.

When designing a database schema and implementing the corresponding ORMs, it's helpful to first sketch out the structure of the data (Figure 6.2). Start by identifying the key objects (which map to database tables), their fields, and their relationships.

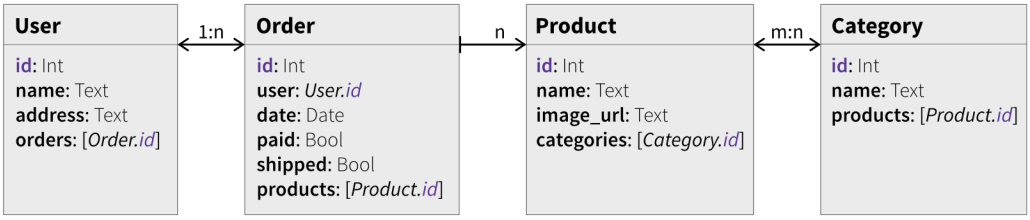


Figure 6.2.: The classes `User`, `Order`, `Product`, and `Category` are mapped to the corresponding tables `users`, `orders`, `products`, and `categories`. Every record in a table is uniquely identified by a **primary key**, often named `id`. Fields in a table can either store data directly (e.g., text or boolean values) or reference records in another table, establishing **relationships between tables**. These relationships are defined using **foreign keys**, which store the primary key of a related record. For example, an `Order` references a single `User` ID (indicating the user who placed the order) and multiple `Product` IDs (the items included in the order).
Relationships between tables can be **bidirectional** or **unidirectional**, depending on the use case. For instance, when querying a `Product`, we want to list all the categories it belongs to, and vice versa. In contrast, the relationship between `Order` and `Product` only goes one way: when retrieving an `Order`, we want to know which products are included, but querying a `Product` doesn't usually require listing all the orders it appears in.

Database migrations, or schema changes, often require careful coordination between code and database updates. For instance, renaming a field means you have to update your database and modify

the code accessing it at the same time. Keeping migration scripts and application code (including ORMs) in the same repository helps ensure consistency during such updates.

> Managing databases in Python
>
> The SQLModel library is highly recommended when working with relational databases in Python and also includes a great tutorial to learn more about ORMs and databases in general. For database migrations, check out Alembic.

APIs

In contrast to a user interface, through which a human interacts with a program, an **Application Programming Interface (API)** enables software components to communicate with one another. Think of it as a contract that defines how different systems interact. For example, an API might specify the classes and functions a library provides so developers can integrate it effectively.

APIs are often associated with **Web APIs**, which provide functionality over the internet. These can either be external services, like the Google Maps API for retrieving directions, or a custom-built **backend** that serves as an **abstraction layer for a database**. This abstraction is useful because it can combine data, enforce rules (e.g., verifying user permissions), and maintain a consistent interface even when the database structure changes.

Interacting with APIs

Web APIs typically use four HTTP methods that correspond to the CRUD (Create, Read, Update, Delete) operations in databases:

- **GET**: Retrieves data, most commonly used when accessing websites. You can include additional parameters by appending a `?` to the URL. For example, `https://www.google.com/search?q=web+api` searches for "web api" using the query parameter `q`. To pass multiple parameters, separate them with `&`.
- **POST**: Sends data to create a new record, often as a JSON object, for example, when submitting a form.
- **PUT**: Updates an existing record.
- **DELETE**: Removes a record.

As a user, you typically interact with APIs through a website's **frontend**, which triggers these API calls in the background. However, APIs can also be queried directly to access raw data, usually returned in **JSON** format.

> API keys and authentication
>
> Many APIs require an **API key** to access their functionality. This key serves as an identifier, allowing the API to authenticate users, track usage, and apply rate limits to prevent abuse. Always keep your API keys secure and avoid exposing them in public repositories or client-side code.

There are many free public APIs you can explore. As an example, we'll use The Cat API to demonstrate how to interact with an API.

You can perform a **GET request** directly in your web browser. For instance, by visiting `https://api.thecatapi.com/v1/images/search?limit=10`, you'll receive a JSON response containing a list of 10 random cat image URLs along with additional details like the image IDs.

For more advanced requests, such as **POST**, you'll need specialized tools. Popular GUI clients include Postman, Insomnia, and Bruno. If you prefer command-line tools, `curl` is a powerful option. Alternatively, you can interact with APIs programmatically using your preferred programming language and relevant libraries.

Interacting with APIs programmatically

In the examples below, we use `curl` and Python to interact with The Cat API to retrieve the latest votes for cat images with a GET request and submit a new vote using a POST request.

Using `curl`

Ensure `curl` is installed by running `which curl` in a terminal—this should return a valid path to your installation.

```
# GET request to view the last 10 votes for cat images
curl "https://api.thecatapi.com/v1/votes?limit=10&order=DESC" \
-H "x-api-key: DEMO-API-KEY"

# POST request to submit a new vote
# the payload after -d is the JSON object submitted to the API
curl -X POST "https://api.thecatapi.com/v1/votes" \
-H "Content-Type: application/json" \
-H "x-api-key: DEMO-API-KEY" \
-d '{
  "image_id": "HT902S6ra",
  "sub_id": "my-user-1234",
  "value": 1
}'
# now run the GET request again to see your new vote
```

Using Python

Python's `requests` library is great for working with APIs.

```python
import requests

BASE_URL = "https://api.thecatapi.com/v1/votes"
API_KEY = "DEMO-API-KEY"

# GET request to fetch the last 10 votes
def get_last_votes():
    response = requests.get(
        BASE_URL,
        headers={"x-api-key": API_KEY},
        params={"limit": 10, "order": "DESC"}
    )
    if response.status_code == 200:
        print(response.json())
    else:
        print(f"Error: {response.status_code}")

# POST request to submit a new vote
def submit_vote(image_id, sub_id, value):
    data = {"image_id": image_id, "sub_id": sub_id, "value": value}
    response = requests.post(
        BASE_URL,
        headers={"Content-Type": "application/json", "x-api-key": API_KEY},
        json=data
    )
    if response.status_code == 201:
        print("Vote submitted!")
    else:
        print(f"Error: {response.status_code}")

if __name__ == '__main__':
    get_last_votes()
    submit_vote("HT902S6ra", "my-user-1234", 1)
```

Implementing APIs

When designing an API, specifically a REST (REpresentational State Transfer) API, it's important to understand the concept of an **endpoint**. An endpoint is a specific URL in your API where a resource can be accessed or modified. For example, if you're building a photo-sharing app, an endpoint like `/images` might allow users to view or upload images. Endpoints should be named using descriptive, plural nouns (e.g., `/users`, `/images`) to clearly represent the resources being accessed (possibly corresponding directly to your database tables). It's also best practice to avoid including verbs in endpoint names (e.g., `/get_users`), since the HTTP method (like GET or POST) already specifies the action being taken, such as retrieving or creating data.

Another key design principle is **statelessness**. Similar to the concept of pure functions, this means

that each API request should contain all the information needed to complete the action, like user authentication tokens. This way, the server doesn't need to remember anything about previous requests, making the API easier to scale. This is especially important in cloud-based environments where multiple requests from the same user may be routed to different servers [5].
Data that needs to be persisted can be stored either in the frontend (client) or a central database in the backend (server), depending on its purpose. Temporary data, like a shopping cart, can be maintained on the user's machine using cookies or local storage. Permanent data, such as a purchase order, is best stored in the backend database to ensure long-term accessibility. This approach supports stateless APIs, as the backend server doesn't need to keep the session state in memory. Instead, all necessary data is either included in the request or can be fetched from the database, allowing each request to be processed independently.

> **Implementing APIs in Python with `FastAPI`**
>
> FastAPI is a Python framework that makes building APIs straightforward. With just a few lines of code, you can turn a function into an endpoint that validates input and returns a JSON response. It's beginner-friendly and highly performant.

Communicating with a Server: Use Cases

Code that runs on our **local devices** (like your laptop or smartphone) is often **limited by the available hardware**. For example, if you play a chess game on your phone against a bot, this bot will only have limited capabilities since the processor on your phone is not sufficient to run an advanced AI model (Figure 6.3).

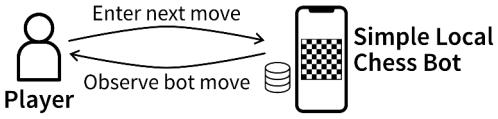


Figure 6.3.: The user interacts only with their device to play chess against a simple bot locally on their phone. The small database icon next to the phone symbolizes the local storage on your phone, where the current game state is saved after each move in case the app is closed so that you can retrieve this saved state the next time you open the app and continue with the game.

A **server in the cloud**, on the other hand, has **the necessary hardware** (in this case multiple GPUs) to run an advanced AI-based chess bot (Figure 6.4). However, to play against this more challenging opponent, you need to have internet access to be able to submit your current game state to the server and receive the next bot move as a response. This also means that **you can't play the game in case your internet connection is not sufficient or the server is down.**

Finally, **by communicating with a server we can also play against other human players** (Figure 6.5). For this, both players take turns in submitting and receiving their next moves to the server. Since the chess API may run on multiple server nodes to handle a high load of requests, it can't be guaranteed that both players will interact with the same node, i.e., we need a stateless design.

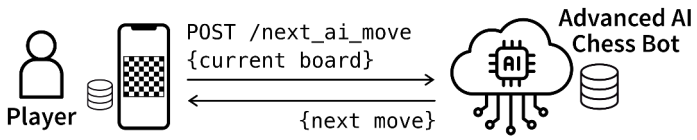


Figure 6.4.: The user interacts with their device and data is submitted to and received from a server in the cloud hosting an advanced AI to determine the next bot move.

To accomplish this, the server persists and retrieves data in a central database that can be accessed by all nodes.[2]

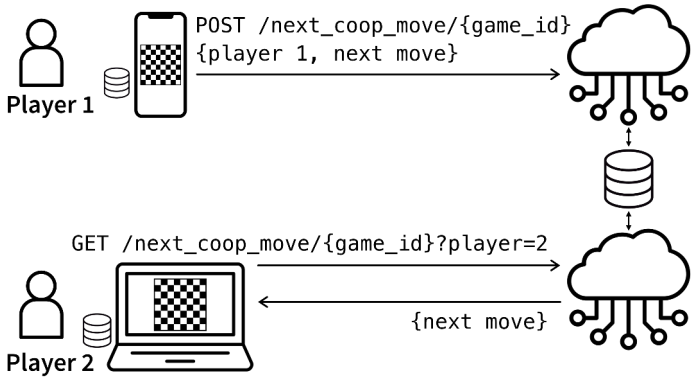


Figure 6.5.: Player 1 interacts with their device and data about the next move is submitted to a server node, where it is stored in a central database. The device from player 2 requests an update from a different server node, which accesses the same database to retrieve the move submitted by player 1.

Asynchronous Communication

When your script calls a library function or API and waits for it to return before continuing with the rest of the code, this is an example of **synchronous communication**. It's similar to a conversation where one person speaks and then waits and listens while the other person responds.

In contrast, **asynchronous (async) communication** allows the program to **keep running while waiting for a response**. Once the response arrives, it is processed and integrated into the workflow, but until then the code just continues without it. Just like when you send an email to someone asking for some data and they send you the results a few hours later.
For example, a website might fetch data from multiple APIs, showing placeholders until the responses arrive. This approach improves the user experience because it keeps the user interface (UI) responsive and enables faster loading by **processing multiple tasks in parallel**.

[2]This use case could also be implemented using an event-driven architecture as discussed in the next section. With this setup, both players would publish their next moves to a message queue and subscribe to it to receive updates about the moves of their opponent.

Event-Driven Architecture

For most applications, communicating directly with external services—whether synchronously or async—is the right approach, because eventually the requested data is needed to finish the original task. But there are also use cases where it's enough that your message was received and you don't need to wait for a response. For example, when placing an order in an online shop, users only care that the order was submitted successfully. They don't wait in front of the screen until it was packaged and shipped—which could take days. An email notification can inform them of progress later.

Such a scenario calls for an **event-driven architecture**, which takes async communication to the extreme. Here, multiple services can operate independently by exchanging information via events using a **message queue (MQ)**, a system that temporarily stores event messages like JSON documents. These messages act as instructions, containing all relevant details about an event, such as a user's order information. Publishers (senders) create events, and subscribers (receivers) process them based on their type, such as `Order Submitted`.

An event-driven architecture offers several **advantages**. By decoupling components, it allows publishers and subscribers to run independently, even in different programming languages or environments. This makes it easier to scale systems and assign teams to own specific components without needing to understand the full system. Additionally, one event can trigger multiple actions. For instance, when an order is packed, one system might update the user database while another generates a shipping label. This approach thereby simplifies the propagation of data to multiple services and can also facilitate replication of live data to testing and staging environments.

However, this type of architecture also brings with it some **challenges**. Since no single component has a full view of the system, tracking the state of a specific task, such as whether an order is still waiting or in progress, can be difficult. Furthermore, while MQs often guarantee that each message is handled at least once, the system requires careful design in case a message is processed multiple times. For example, if a subscriber crashes after processing a message but before confirming its completion, the MQ might reassign the task to another instance, potentially leading to duplicate processing. For these reasons, event-driven architectures should only be used when direct communication between services is not an option [28].

Batch Jobs

Unlike continuously running services such as web APIs, **batch jobs** are scripts or workflows used to process accumulated data in one go. They are particularly effective when tasks don't require immediate processing or when grouping tasks can improve efficiency. To automate recurring tasks, batch workflows can be scheduled at specific intervals using tools like **cron jobs**.[3]

Examples of scenarios where batch jobs are useful include:

- Fetching new messages from a queue every 10 minutes to process them in bulk, reducing overhead.
- Generating a sales report for the marketing department every Monday at midnight.
- Running nightly data validation to check for data drift or anomalous patterns.
- Retraining a machine learning model every week using newly collected data to create updated recommendations, like Spotify's "Discover Weekly" playlist [15].

[3]Tools like Crontab Guru can help configure these schedules.

For large-scale jobs, distributed systems might be necessary to ensure they complete within an acceptable timeframe.

Software Design Revisited

As your software evolves beyond a simple script, it becomes a system composed of multiple interconnected components. Each component can be viewed as a subsystem with its own defined boundaries and interface, responsible for specific functionalities while interacting with other parts of the system.

To manage this growing complexity, it's best to think of these components—such as a **GUI/client/frontend**, **API/server/backend**, and **data storage** (files, databases, message queues)—as distinct layers. A clean design follows the principle of **layered communication**, where each layer interacts only with the layer directly below it [22]. For example, the client communicates with the server, and the server interacts with the database, avoiding "skip connections" where one layer bypasses another.
This design principle minimizes dependencies and makes the system easier to maintain: If the interface of one component changes, only the layer directly above it has to be adapted, for example, if a field in the database is renamed, only the API needs to be updated.

When you design these more complicated systems, it's even more important to sketch the overall architecture before you start with the implementation (Figure 6.6). Visualizing how the layers interact can reveal potential bottlenecks or unnecessary complexity and gives you and your collaborators clarity on the big picture.

Begin by visualizing the flow from the user's perspective, focusing on how they interact with the client to complete their intended task. Once the user experience is clear, **continue with the server and data store layers** and map out how these can support and implement each step:

- What data is needed to render the GUI?
- What data needs to be persisted in the database for later?

Finally, in accordance with the read and write (CRUD) operations on the database, you can **design the ORMs** used to represent and store the required data (Figure 6.2).

Domain-Driven Design

As a software product grows, it eventually becomes too complex for a single team to manage. To reduce cognitive load and improve maintainability, the system should be broken into smaller **subsystems**, each of which can be **owned and managed autonomously** by a dedicated team.

To minimize inter-team dependencies and reduce communication overhead, it's best to split the system along **domain or use-case boundaries**, making a cut through all technical layers [18][23]. For example, in an e-commerce platform, one team might own the "search & browse" domain, while another handles "purchasing." Although implementing a new feature may involve both client and server changes, those changes typically stay within a single domain, allowing the responsible team to implement them independently.

Technically, this often means breaking a monolith into **domain-based microservices** [23], where each service operates independently (which also means it needs its own database). The key challenge

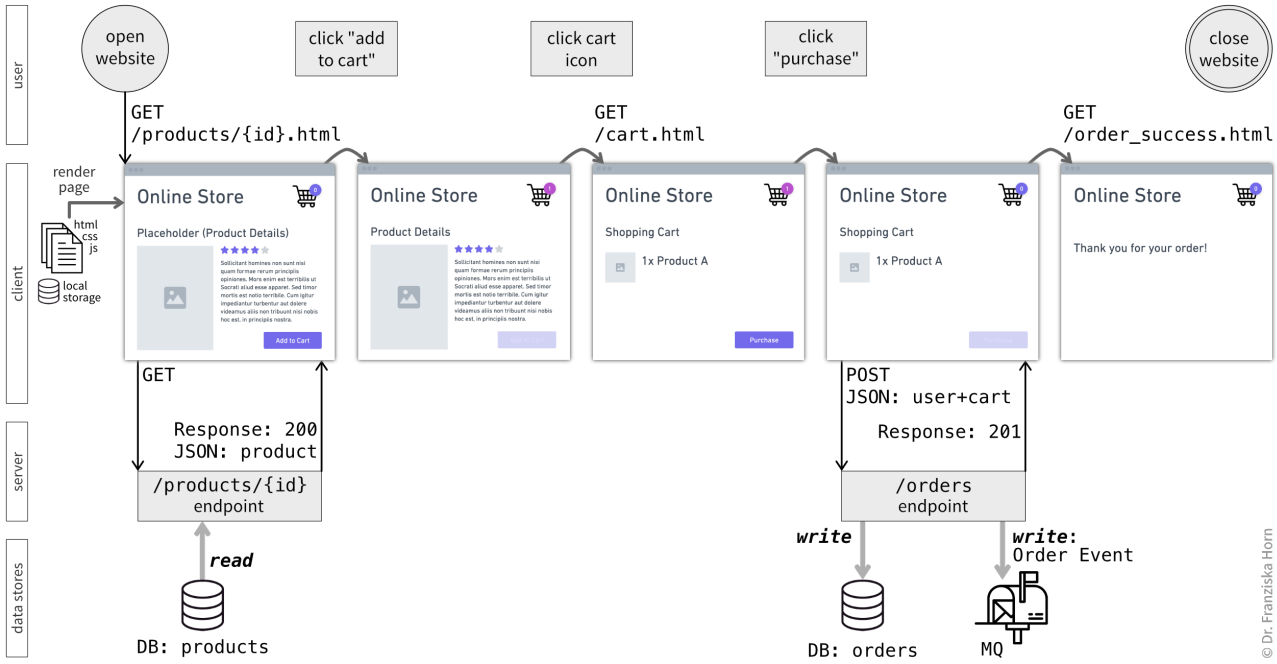


Figure 6.6.: A simplified flow showing what happens when a user orders something online: the user opens a product page, which triggers a request to the server to fetch the corresponding product details from the database (DB). The user then clicks the "add to cart" button, which places the product into the shopping cart (in local storage managed by the client). The user then views the shopping cart and clicks "purchase", which triggers a POST request to the API, submitting the user's cart contents. The server creates a new record in the `orders` table to store the purchase details and submits an `Order` event to the message queue (MQ), thereby alerting other services that a new order needs to be packed and shipped. The endpoint returns with the status code 201 ("success") and the client redirects the user to a page that tells them the purchase was successful, at which point the user closes the tab.

is balancing **local vs. global complexity**: Each microservice should be **small enough** to be fully understood and maintained by one team, but **large enough** to encapsulate meaningful logic. If services are too granular, they require excessive inter-service communication to get anything done, leading to tightly coupled systems, only now with the added complexity of a distributed system and multiple deployments.

6.2. Delivery & Deployment

Modern software development requires reliable and efficient processes to build, test, and deploy applications [9]. Delivery and deployment strategies ensure that new features and updates are released quickly, safely, and at scale, minimizing disruptions to users while maintaining quality.

CI/CD Pipelines

Continuous Integration (CI) and **Continuous Delivery/Deployment (CD)** pipelines are the backbone of modern software practices. CI focuses on automating the process of integrating code changes into a shared repository. Every change triggers automated tests to ensure that the new code works harmoniously with the existing codebase. CD extends this by automating the preparation or deployment of changes into production, either ready for manual approval (Continuous Delivery) or fully automated (Continuous Deployment). This drastically reduces manual effort, minimizes human error, and enables faster iteration cycles.

CI/CD pipelines are either included directly into version control platforms, such as GitHub Actions and GitLab CI/CD, or can be run using external tools like Jenkins or CircleCI.

Optimizing CI/CD Pipelines

To enhance pipeline efficiency and reliability, consider the following practices:

- **Dependency caching:** Cache dependencies to reduce the time spent downloading and installing them for each build.
- **Selective testing:** Run only the tests affected by recent changes to speed up feedback.
- **Real-time notifications:** Notify developers immediately when a pipeline fails, enabling faster issue resolution.

> **!** Security in CI/CD pipelines
>
> Security must be a priority in any CI/CD process. For example, it is best practice to include a **dependency scanning** step to detect vulnerabilities in third-party libraries. Furthermore, you should never include **sensitive information**—such as API tokens, database credentials, or private keys—directly in your code. However, because CI jobs often require access to this information, you can securely store secrets using dedicated CI/CD variables or external **secret management tools** like HashiCorp Vault or AWS Secrets Manager.

A well-designed CI/CD pipeline not only saves time and resources but also ensures a consistent and high-quality delivery of software.

Containers in the Cloud

Containers, powered by tools like **Docker**, encapsulate applications with their dependencies, ensuring consistency across different environments. This portability simplifies deployment and reduces issues caused by environment differences.

For managing containerized applications at scale, **Kubernetes (k8s)** is the industry standard. Kubernetes automates the **orchestration of containers**, providing features like:

- **Auto-scaling:** Adjust resources dynamically based on workload.
- **Self-healing:** Automatically restart failed containers.
- **Load balancing:** Distribute traffic efficiently across services.

Using Cloud Platforms

Cloud platforms like **AWS**, **Google Cloud Platform (GCP)**, and **Microsoft Azure** offer robust infrastructures for deploying and scaling applications. For simpler workflows, **managed services** like Render or Heroku abstract away much of the operational complexity.

Managing costs effectively is critical in cloud deployments. Key strategies include:

- **Resource scaling:** Reduce unused resources during off-peak hours.
- **Serverless computing:** Use serverless models, like AWS Lambda, for infrequent workloads to save costs.
- **Cost monitoring tools:** Leverage AWS Cost Explorer or GCP Billing to track and optimize spending.

Infrastructure as Code (IaC)

Instead of configuring your cloud setup manually through the platform's GUI, it is highly recommended to use **Infrastructure as Code** tools like Terraform and AWS CloudFormation to manage cloud infrastructure programmatically. The IaC configuration files can then be version-controlled, which ensures:

- Reproducible setups for consistent environments.
- Easier onboarding for new team members.
- Reduced risk of configuration drift.

Testing and Staging Environments

Deploying changes directly to production is risky. To ensure stability:

- Use **staging environments** that mimic production to validate changes before release.
- Maintain **testing environments** for early experimentation and debugging.

Techniques like **A/B testing** and **feature flags** allow gradual rollouts or controlled exposure of new features, minimizing user disruption. This can be achieved using deployment strategies like:

- **Blue-green deployments:** Maintain two environments (blue and green) and switch traffic between them for A/B tests or to reduce downtime during updates.
- **Canary releases:** Gradually expose updates to a small group of users, monitoring for issues before full deployment.

Scaling Considerations

As applications grow, scaling requires thoughtful architectural design. You should consider:

- **Task separation:** For example, train machine learning models periodically as batch jobs, while keeping prediction services running continuously. This is particularly important when services have vastly **different user bases** (e.g., hundreds of admins versus millions of regular users), as they require varying replication rates for horizontal scaling. Especially if services rely on distinct dependencies, combining them into a single Docker container can result in a large, inefficient image, which increases the services' startup time.
- **Team autonomy:** Design services such that teams can own and work on individual components independently, thereby reducing communication overhead and speeding up development cycles [30].

Monitoring and Observability

To ensure smooth operation and detect issues proactively, monitoring and observability are essential. Focus on:

- **System performance:** Monitor the "golden signals"—latency, traffic, errors, and saturation of your services. Tools like Prometheus and Grafana are commonly used for this.
- **Data quality:** Track changes in input data distributions and monitor metrics like model accuracy to detect data drift.
- **Synthetic monitoring:** Simulate user behavior to identify bottlenecks and improve responsiveness. Complement this with chaos engineering tools like **Chaos Monkey** to test your system's resilience by deliberately introducing failures, ensuring your infrastructure can handle unexpected disruptions effectively.
- **Distributed tracing:** Debug across microservices using tools like Jaeger or OpenTelemetry.

When issues arise, having a rollback strategy is crucial. Options include:

- Reverting to a stable container image.
- Rolling back database migrations.
- Using feature flags to disable problematic updates.

By combining robust delivery pipelines, thoughtful architecture, and effective monitoring, teams can ensure that their applications remain reliable, scalable, and adaptable to changing needs.

> Before you continue
>
> At this point, you should have a clear understanding of:
>
> - Which additional steps you could take to make your research project production-ready.

Afterword

You're still at the beginning of your journey towards professional software engineering. But I hope this book could give you a glimpse into what lies ahead.

If you want to learn more, I recommend these for your next read:

- [32] *The Pragmatic Programmer: Your Journey to Mastery* by David Thomas and Andrew Hunt (20th Anniversary Edition, 2019) – This book expands on many of the coding principles we've touched on, offering practical advice for succeeding as a programmer in industry.
- [31] *The Algorithm Design Manual* by Steven Skiena (2020) – A deep dive into algorithms and data structures to sharpen your algorithmic thinking skills.

Iterate Faster

Research is an inherently iterative process (Figure 6.7): You start with an exciting hypothesis, run experiments, face disappointing results, reflect, learn, refine your ideas, and eventually gain enough insight to publish.

I believe that Clarity-Driven Development (CDD) will help you accelerate this cycle. It enables you to form stronger hypotheses and implement experiments more efficiently. Disappointing results will still happen—there's no avoiding that—but with CDD, your iteration loops become shorter, and progress comes faster.

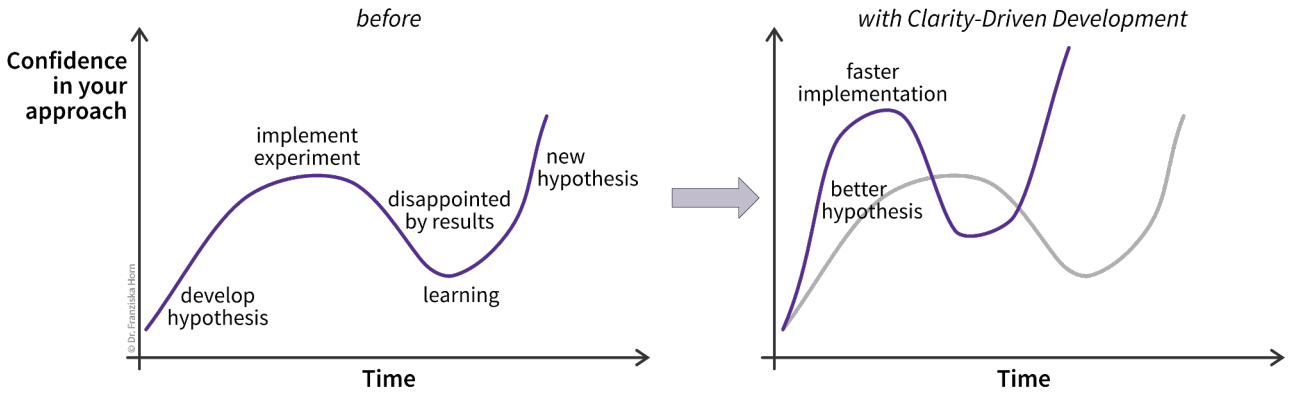


Figure 6.7.: Research is often an emotional roller coaster. With CDD you go higher, faster.

Good luck!

Give Me Feedback, Please!

I'm always looking to improve the contents of this book (or any other resources you can find on my website). Therefore, I would be eternally **grateful for your feedback**:

- How has this book changed your approach to programming?
- What worked for you and where did you get stuck?
- What is this book missing?

Please send me an email to `hey@franziskahorn.de` and let me know what you think!

References

[1] Aulet B. *Disciplined Entrepreneurship: 24 Steps to a Successful Startup, Expanded & Updated*. John Wiley & Sons (2024).

[2] Beck K. *Tidy First?* O'Reilly Media, Inc. (2023).

[3] Callaway E. Chemistry Nobel Goes to Developers of AlphaFold AI That Predicts Protein Structures. *Nature* 634(8034), 525–526 (2024).

[4] Christiansen J. *Building Science Graphics: An Illustrated Guide to Communicating Science Through Diagrams and Visualizations*. AK Peters/CRC Press (2022).

[5] Davis C. *Cloud Native Patterns: Designing Change-Tolerant Software*. Manning Publications (2019).

[6] Dean J, Ghemawat S. MapReduce: Simplified Data Processing on Large Clusters. *Communications of the ACM* 51(1), 107–113 (2008).

[7] Foote B, Yoder J. Big Ball of Mud. *Pattern languages of program design* 4, 654–692 (1997).

[8] Ford N, Parsons R, Kua P, Sadalage P. *Building Evolutionary Architectures*. O'Reilly Media, Inc. (2022).

[9] Forsgren N, Humble J, Kim G. *Accelerate: The Science of Lean Software and Devops: Building and Scaling High Performing Technology Organizations*. IT Revolution (2018).

[10] Fowler M. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Professional (2018).

[11] Freiesleben T, Molnar C. *Supervised Machine Learning for Science: How to Stop Worrying and Love Your Black Box*. (2024).

[12] Helmers L, Horn F, Biegler F, Oppermann T, Müller K-R. Automating the Search for a Patent's Prior Art with a Full Text Similarity Search. *PLoS ONE* 14(3), e0212103 (2019).

[13] Horn F. *A Practitioner's Guide to Machine Learning*. (2021).

[14] Horn F. *Similarity Encoder: A Neural Network Architecture for Learning Similarity Preserving Embeddings*. Technische Universitaet Berlin (Germany) (2020).

[15] Huyen C. *Designing Machine Learning Systems*. O'Reilly Media, Inc. (2022).

[16] Irving D, Hertweck K, Johnston L, Ostblom J, Wickham C, Wilson G. *Research Software Engineering with Python*. (2021).

[17] Khononov V. *Balancing Coupling in Software Design: Universal Design Principles for Architecting Modular Software Systems*. Addison-Wesley Professional (2024).

[18] Khononov V. *Learning Domain-Driven Design*. O'Reilly Media, Inc. (2021).

[19] Kleppmann M. *Designing Data-Intensive Applications: The Big Ideas Behind Reliable, Scalable, and Maintainable Systems*. O'Reilly Media, Inc. (2017).

[20] Knaflic CN. *Storytelling with Data: A Data Visualization Guide for Business Professionals*. John Wiley & Sons (2015).

[21] Lemaire M. *Refactoring at Scale*. O'Reilly Media, Inc. (2020).

[22] Lilienthal C. *Sustainable Software Architecture: Analyze and Reduce Technical Debt*. dpunkt.verlag (2019).

[23] Lilienthal C, Schwentner H. *Domain-Driven Transformation: Monolithen Und Microservices Zukunftsfähig Machen*. dpunkt.verlag (2023).

[24] McChrystal GS, Collins T, Silverman D, Fussell C. *Team of Teams: New Rules of Engagement for a Complex World*. Penguin (2015).

[25] Normand E. *Grokking Simplicity: Taming Complex Software with Functional Thinking*. Manning Publications (2021).

[26] Ousterhout JK. *A Philosophy of Software Design*. Yaknyam Press Palo Alto, CA, USA (2018).

[27] Page-Jones M. *What Every Programmer Should Know about Object-Oriented Design*. Dorset House (1995).

[28] Richards M, Ford N. *Fundamentals of Software Architecture: An Engineering Approach*. O'Reilly Media, Inc. (2020).

[29] Serra J. *Deciphering Data Architectures*. O'Reilly Media, Inc. (2024).

[30] Skelton M, Pais M. *Team Topologies: Organizing Business and Technology Teams for Fast Flow*. IT Revolution (2019).

[31] Skiena SS. *The Algorithm Design Manual*. Springer (2020).

[32] Thomas D, Hunt A. *The Pragmatic Programmer: Your Journey to Mastery, 20th Anniversary Edition*. Addison-Wesley Professional (2019).