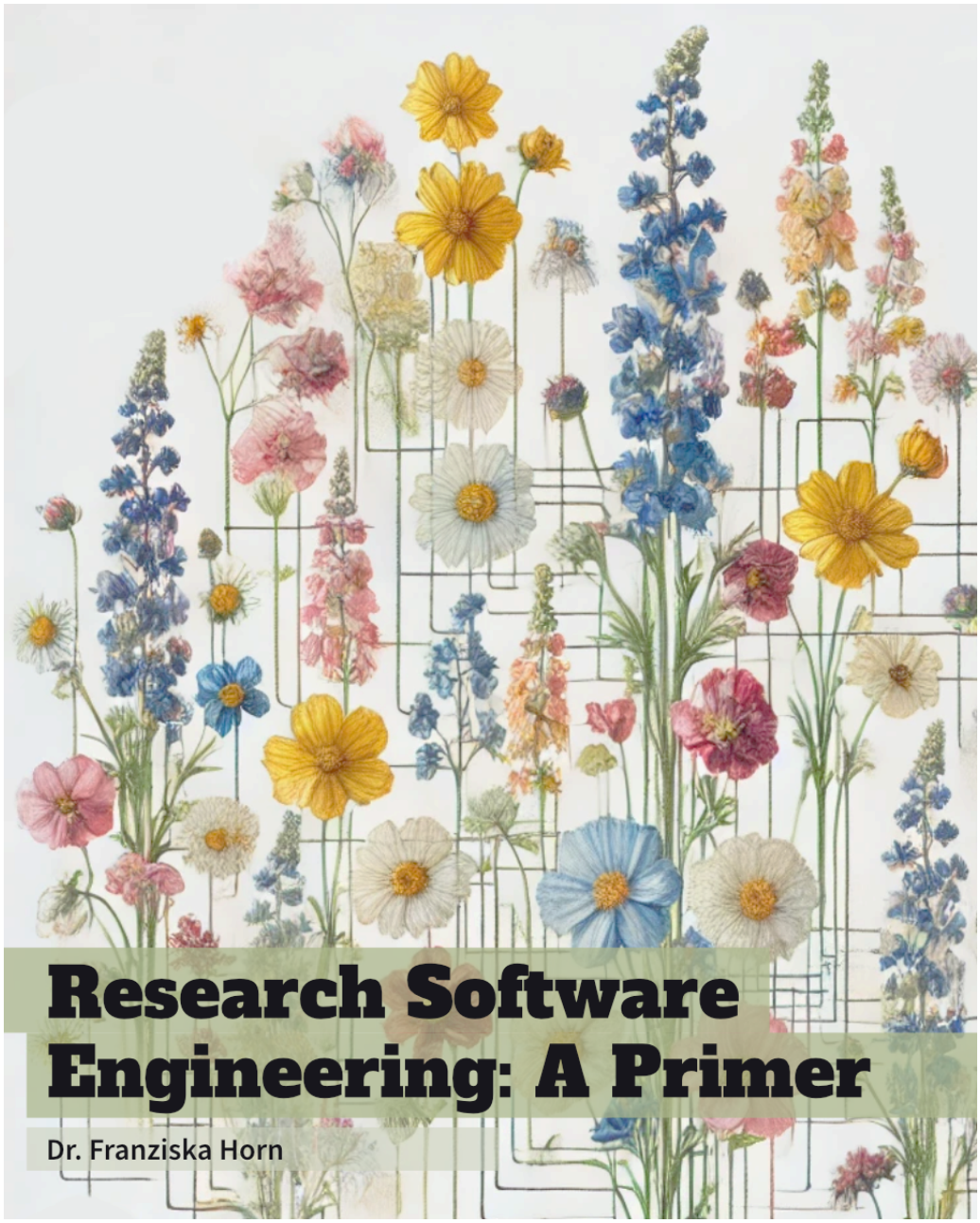


Research Software Engineering: A Primer

Dr. Franziska Horn

2025-01-01

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 Biologists* and can put together functional scripts through trial and error (with a little help from ChatGPT). 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.

This is still a draft version! Please write me an email, if you have any suggestions for how this book could be improved!

Enjoy!

Acknowledgments

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

How to cite

```
@book{horn2025rseprimer,
  author = {Horn, Franziska},
  title = {Research Software Engineering: A Primer},
  year = {2025},
  url = {https://franziskahorn.de/rsebook/},
}
```

1 Research Purpose

Before writing your first line of code, it's crucial to have a clear understanding of what you're trying to achieve—specifically, the purpose of your research. This clarity will not only help you reach your desired outcomes more efficiently but will also be invaluable when collaborating with others. Being able to explain your goals effectively ensures everyone is aligned and working toward the same objective.

We'll begin with an overview of common research goals and the types of data analysis needed to achieve them. Then, we'll discuss how to quantify the outcomes you're trying to achieve. Finally, we'll explore how to visually communicate your research purpose, as visual representations are often the most effective way to convey complex ideas.

1.1 Types of Research Questions

In research, your goal is to improve the status quo, whether by filling a knowledge gap or developing a new method, material, or process with better properties. Most research questions can be categorized into four broad groups, each associated with a specific type of analytics approach (Figure 1.1).

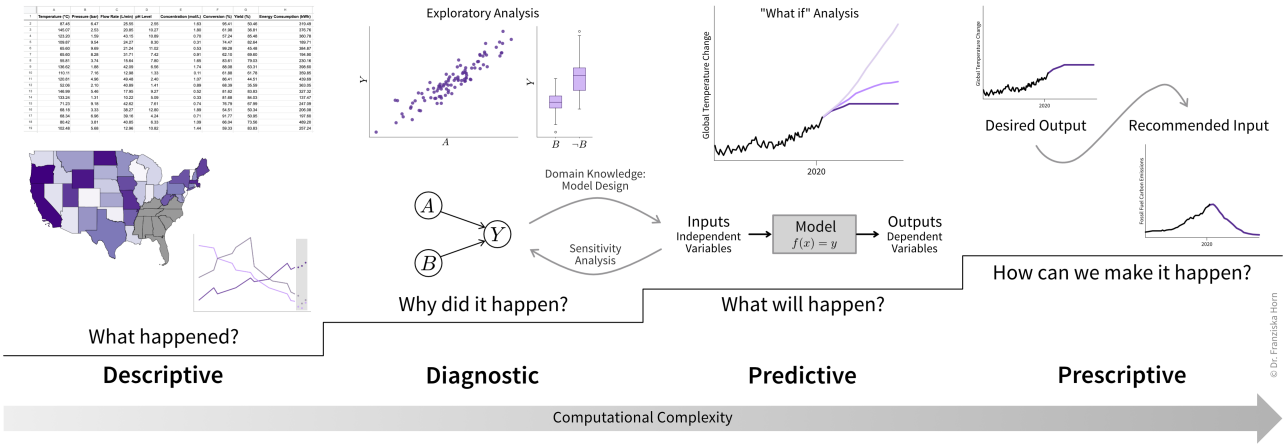


Figure 1.1: Descriptive, diagnostic, predictive, and prescriptive analytics, with increasing computational complexity and need to write custom code.

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, typically using standard software tools.

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 these models match 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 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 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:
  >
  > – [5] Supervised Machine Learning for Science by Christoph Molnar & Timo Freiesleben; A fantastic introduction focused on applying black-box models in scientific research.
  > – [6] A Practitioner's Guide to Machine Learning by me; A broader overview of ML methods for a variety of use cases.

After developing an accurate model, researchers can analyze its behavior (e.g., through a sensitivity analysis, which examines how outputs change with varying inputs) to gain further insights about the system (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.

1.2 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. But you also need to make sure your sample is representative of the population as a whole, i.e., to avoid sampling bias.
- **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):** These are the 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.

Ideally, you should already have an idea of how existing solutions perform on these 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.[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. Clearly defining the key metrics needed to demonstrate your research's impact will help you focus your programming efforts effectively.

1.3 Draw Your Why

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

Visual representations are particularly powerful for conveying complex ideas. One effective approach is creating **"before and after" visuals that contrast the current state of the field with your proposed improvements** (Figure 1.2).

The "before" scenario might show a lack of data, an incomplete understanding of a phenomenon, poor model performance, or an inefficient process or material. The "after" scenario 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, your "after" scenario 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 goal 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 research goals are tied directly to analytics. Sometimes the main improvement is more qualitative, for example, focusing on design or functionality (Figure 1.3). Even in these cases, however, you'll often need to demonstrate that your new approach meets or exceeds existing solutions in terms of other key performance indicators (KPIs), such as energy efficiency, speed, or quality parameters like strength or durability.

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

[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 [2].

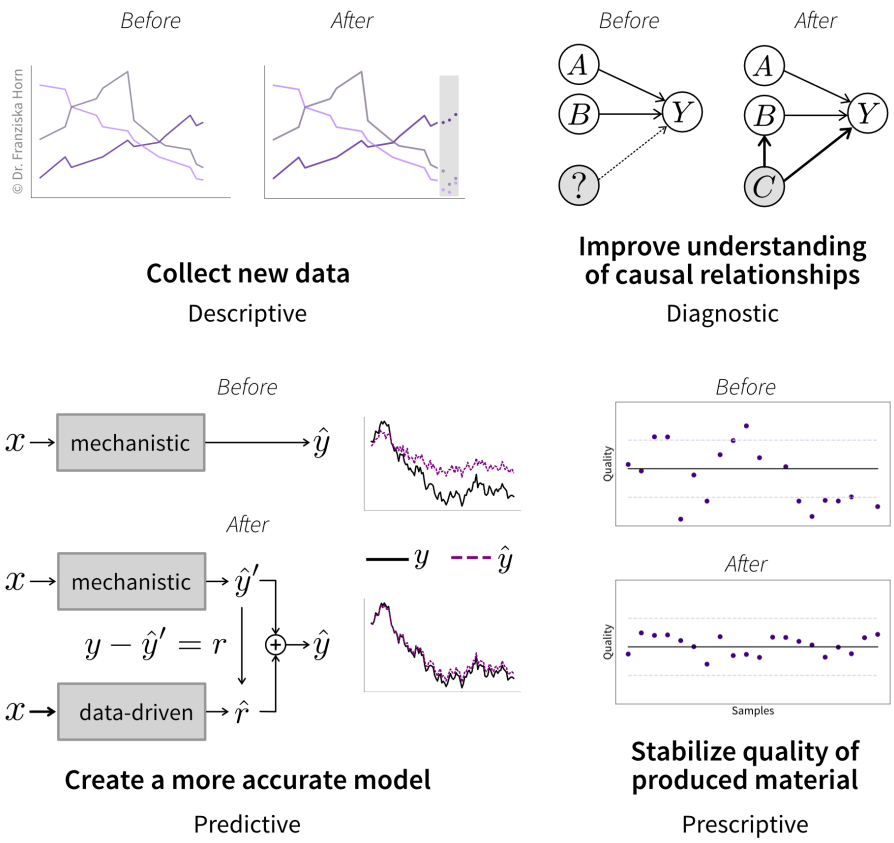


Figure 1.2: Exemplary research goals and corresponding "before and after" visuals for descriptive, diagnostic, predictive, and prescriptive analytics tasks.

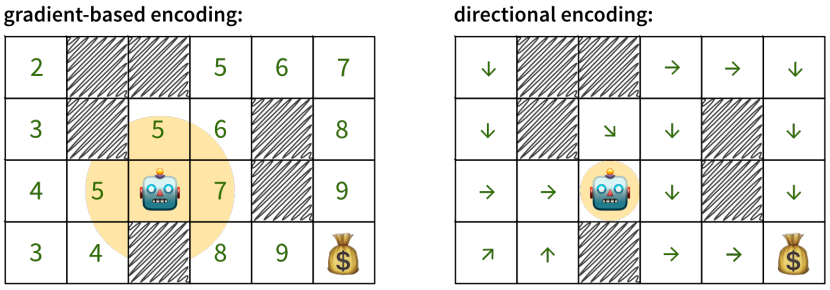


Figure 1.3: This example illustrates a task where a robot must reach its target (represented by money) as efficiently as possible. **Original approach (*left*)**: The robot relied on information encoded in the environment as expected rewards. To determine the shortest path to the target, the robot required a large sensor (shown as a yellow circle) capable of scanning nearby fields to locate the highest reward. **New approach (*right*)**: Instead of relying on reward values scattered across the environment, the optimal direction is now 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 quantitative evaluation:** It still needs to be demonstrated that with the new approach, the robot reaches its target at least as quickly as with the original approach.

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 Data & Results

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.

2.1 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 2.1).

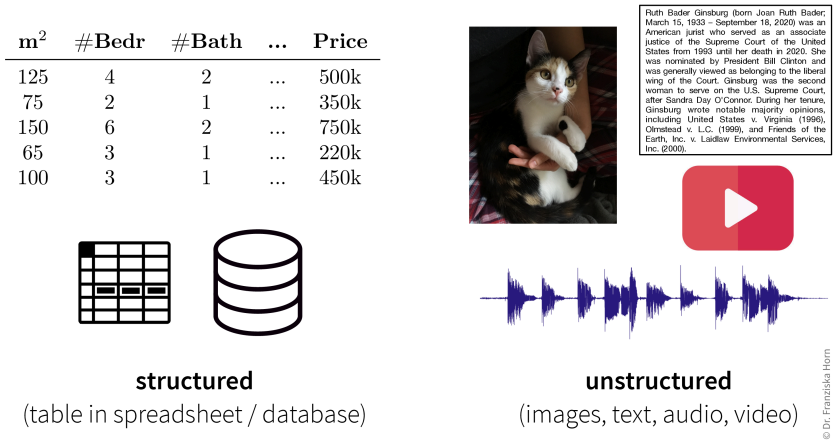


Figure 2.1: Structured and unstructured data.

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).

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.

> **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 arrays or dictionaries containing multiple values, or user-defined objects) and define how information is stored in computer memory.

> **Data types in 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.

2.2 Data Analysis Results

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, 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.

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., in a survey rich people are less likely to answer questions about their income or specific measurements are only collected under certain conditions)? 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**:

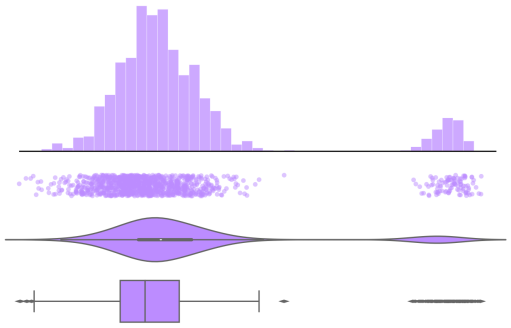


Figure 2.2: Histogram, strip plot, violin plot, box plot, and summary statistics of the same values.

  - 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):

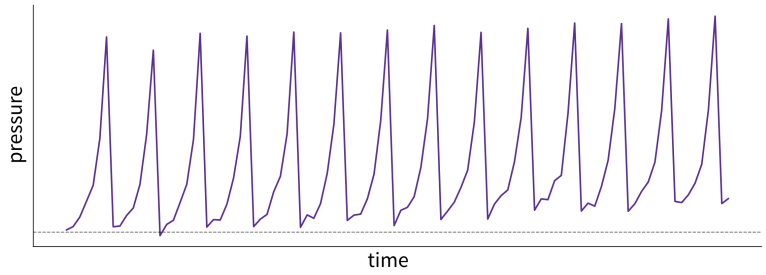


Figure 2.3: What caused these trends and what are their implications for the future? This plot shows fictitious data of the pressure in a pipe affected by fouling—that is, a buildup of unwanted material on the pipe's surface, leading to increased pressure. The pipe is cleaned at regular intervals, causing 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**:

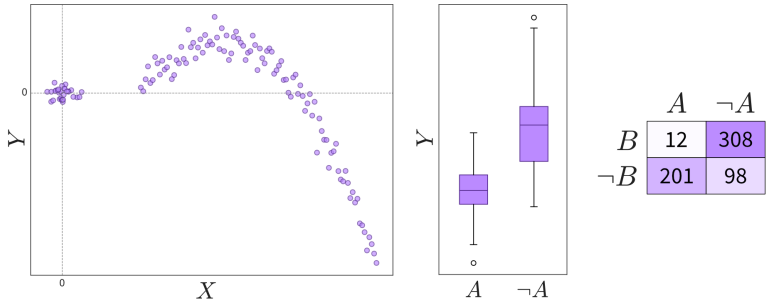


Figure 2.4: Depending on the variables' types (continuous or discrete), relationships can be shown in scatter plots, box plots, or a table. Please note that not all interesting relations between the two variables can be detected through a high correlation coefficient, so you should always check the scatter plot for details.

  - Are the observed correlations between variables expected?

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

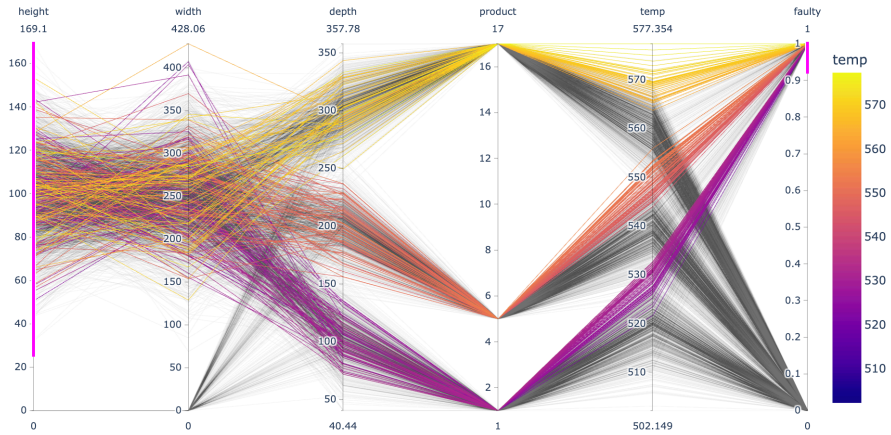


Figure 2.5: Each line in a parallel coordinate plot represents one data point, with the corresponding values for the different variables marked at the respective y-axis. The screenshot here shows an interactive plot created using the Python `plotly` library. By selecting value ranges for the different 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* [7].

> ⚠ 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.

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.

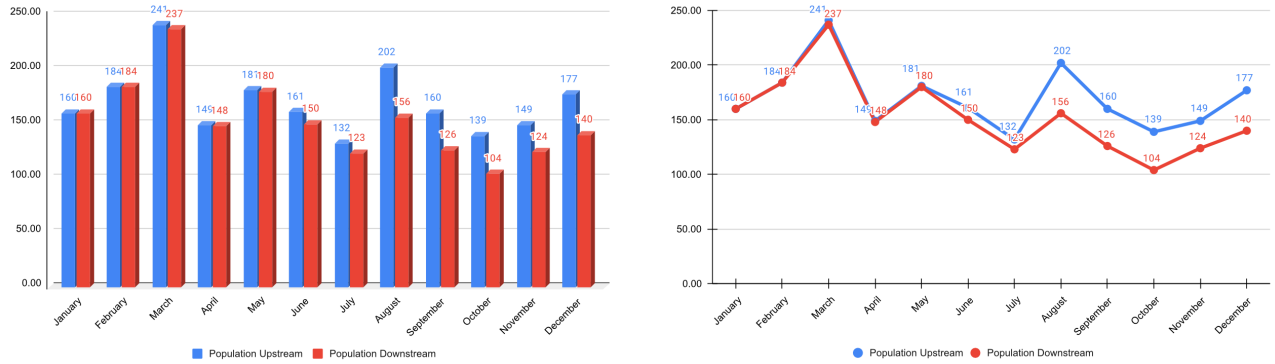


Figure 2.6: *Left:* Bar charts (especially in 3D) make it hard to compare numbers over a longer period of time. *Right:* Trends over time can be more easily detected in line charts. [Example adapted from: *Storytelling with Data* by Cole Nussbaum Knaflic]

Step 2: Cut clutter / maximize data-to-ink ratio

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

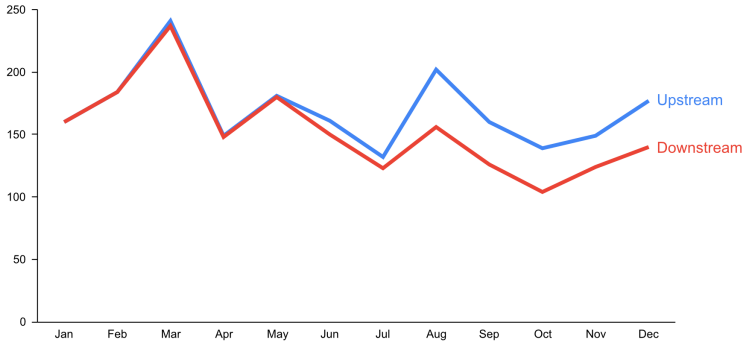


Figure 2.7: Cut clutter! [Example adapted from: *Storytelling with Data* by Cole Nussbaum Knaflic]

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.

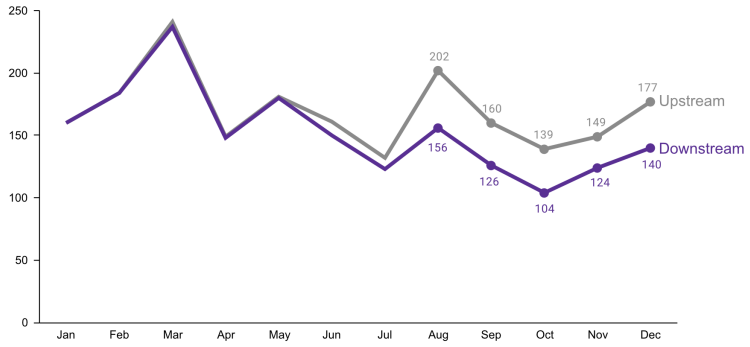


Figure 2.8: Start with gray and use pre-attentive attributes strategically to focus the audience's attention. [Example adapted from: *Storytelling with Data* by Cole Nussbaum Knaflic]

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 drawn 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?").

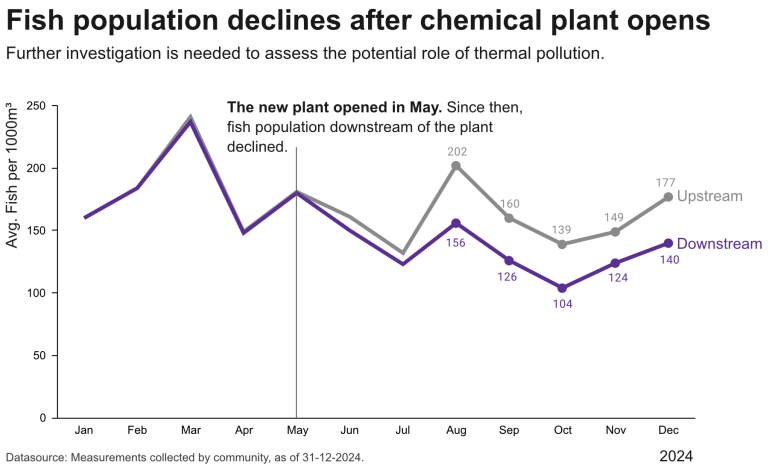


Figure 2.9: Tell a story. [Example adapted from: *Storytelling with Data* by Cole Nussbaum Knaflic]

2.3 Draw Your What

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.

In software development, a UX designer typically creates mockups of a user interface (like the screens of a mobile app) before developers begin coding. Similarly, in our case, we want to start with a clear picture of what the output of our program should look like. The difference is that, instead of users interacting with the software themselves, they'll only see the plots or tables that your program generated, maybe in a journal article.[1]

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. Ask yourself: ***What figures or tables would best communicate the advantages of my approach?***

Depending on your research goals, your results might be as simple as a single number, such as a p-value or the total number of people surveyed. However, if you're reading this, you're likely tackling something that requires a more complex analysis. For example, you might compare your solution's overall performance to several baseline approaches or illustrate how your solution converges over time (Figure 2.10).

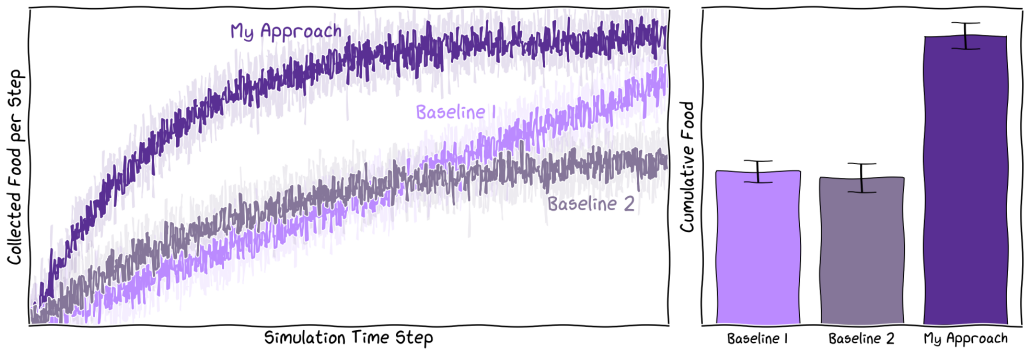


Figure 2.10: Exemplary envisioned results: The plots show the outcome of a multi-agent simulation, where 'my approach' clearly outperforms two baseline methods. In this simulation, a group of agents is tasked with locating a food source and transporting the food back to their home base piece by piece. 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.[2]

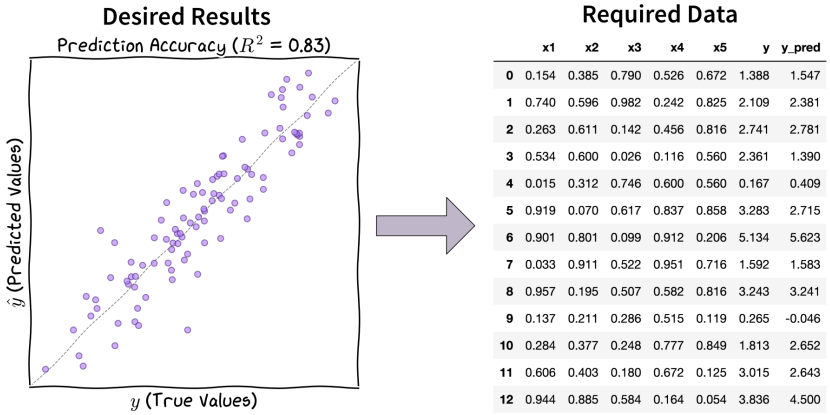
It's important to remember that your actual results might look very different from your initial sketches—they might even show that your solution performs worse than the baseline. This is completely normal. The scientific method is inherently iterative, and unexpected results are often a stepping stone to deeper understanding. By starting with a clear plan, you can generate results more efficiently and quickly pivot to a new hypothesis if needed. 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.

[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. You could even use these sketches of your results as placeholders if you're already drafting a paper or presentation.

[2] These plots and the next were generated with Python using matplotlib's `plt.xkcd()` setting and the xkcd script font. A pen and paper sketch will be sufficient for your case.

Once you've visualized the results you want, **work backward to figure out what data you need** to create them. This is especially important when you're generating the data yourself, such as through simulations. For instance, if you plan to plot how values change over time, you'll need to record variables at every time step rather than just saving the final outcome of a simulation (duh!). Similarly, if you want to report your model's accuracy (Figure 2.11), you'll need:

1. Input variables for each data point to generate predictions (= model output).
2. The actual (true) values for each data point.
3. A way to compute the overall deviation between predictions and true values, such as using an evaluation metric like R^2.



	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 2.11: Work backward from the desired results to determine what data is necessary to create them.

By working backward from your desired results to the required data, you can design your code and analysis pipeline to ensure your program delivers exactly what you need.

> **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.).
> - The underlying data needed to produce these results (e.g., what rows and columns should be in your spreadsheet).

3 Tools

Before we continue with creating your results—i.e., actually start developing software—let's take a quick tour of some tools that can make your software engineering journey smoother.

Although the code examples in this book use Python, the general principles discussed here apply to most programming languages.

3.1 Programming Languages

Different programming languages suit different needs. Here's a quick 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.

3.2 Version Control

Version control is essential in software development to keep track of code changes and collaborate effectively. Think of it as a time machine that lets you revert to any version of your code or examine how it evolved.

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 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, many professionals 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.

3.3 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.
- **Atom**: Open-source and backed by GitHub (though less popular than other tools).
- **Vim** and **Emacs**: Some of the first code editors, often used as command line tools and beloved by keyboard shortcuts enthusiasts.

Full IDEs

For more features, IDEs integrate tools like file browsers, Git support, and debuggers. They are ideal for larger projects and provide support for more complex tasks, like renaming variables across multiple files when you're refactoring your 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.
- **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. We 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 to run your notebook again from top to bottom once you're finished 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.

3.4 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 `poetry`

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

```
[tool.poetry]
name = "example-project"
version = "0.1.0"
description = "A sample Python project"
authors = ["Your Name <youremail@example.com>"]

[tool.poetry.dependencies]
python = "^3.9"
requests = "^2.26.0"  # external libraries incl. versions

[build-system]
requires = ["poetry-core"]
build-backend = "poetry.core.masonry.api"
```

Basic commands:

- `poetry new example-project`: Create a new project (folder incl. `pyproject.toml` file).
- `poetry add [package]`: Add a dependency (can also be done directly in the file).
- `poetry install`: Install all dependencies.
- `poetry shell`: Activate 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 2.10).

> **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.

3.5 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 via `pip` and configure it in the same `pyproject.toml` file that we also used for `poetry`. Then run it over you 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 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).

3.6 Putting It All Together

When you set up all these tools, your repository should now 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.

4 Software Design

Now that your code repository is set up, 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 build**. 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 design the final outcome first. It doesn't have to be perfect—a quick sketch on paper will do. And you'll likely need to adapt as you go (which is fine since we'll design with flexibility in mind!). But the more thought you put into planning, the smoother and faster execution will be.

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

4.1 Avoid Complexity

A common acronym in software engineering is **KISS** - "keep it simple, stupid!". While this may be well-intentioned advice, it can only apply to individual parts of your code, as a full-fledged software is a system that consists of multiple components that interact with each other. Here, the best you can hope for is *complicated*, i.e., to avoid complexity (Figure 4.1).

A **complex** system includes many elements that interact with each other in ways that are not easily traceable or for humans to comprehend. You might change something in one place and suddenly, something far on the other side breaks. That's not good in software. We should always design for change, since inevitably, there is something we need to adapt or extend and when this happens, we want to be confident in what we're doing and not afraid that our change will break something else or have unintended consequences that we're not aware of. No one likes bugs.

This is why we want a **complicated** system: it still includes a lot of elements, but they are grouped in components, neat little subsystems, so the whole can be taken apart and each part can be understood on its own while the whole can also be understood without understanding each individual component in detail. Ideally, this means our system consists of a neat hierarchy of components at different levels of abstraction (Figure 4.2). To understand the code on one level of this hierarchy, we only need to understand what the components one level below are doing to get an idea of the purpose of the code. For example, to understand what is happening in `plot_results`, it is enough to know that there is a scatter plot created and we don't need to know the details of how this plot is created, such as that it requires the computation of R^2. By decomposing code in such a way, we reduce the **cognitive load** that is required to understand what the code is doing.

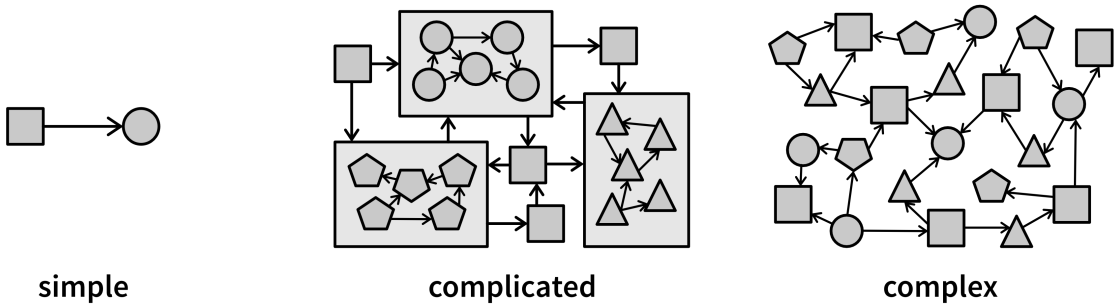


Figure 4.1: (Software) Systems can be of different complexity. A script or function that linearly executes a set of steps could be considered simple. But most software programs are (at least) complicated: they consist of multiple components that interact with each other. However, these can still be broken down into manageable subsystems, which makes it possible to understand the system as a whole. A complex system, in computer science referred to as "spaghetti code" or a "big ball of mud" [4], contains many individual elements and interactions between them – when you change something on one end it is unclear how this will affect the other pieces as it is difficult to understand how all the elements come together.

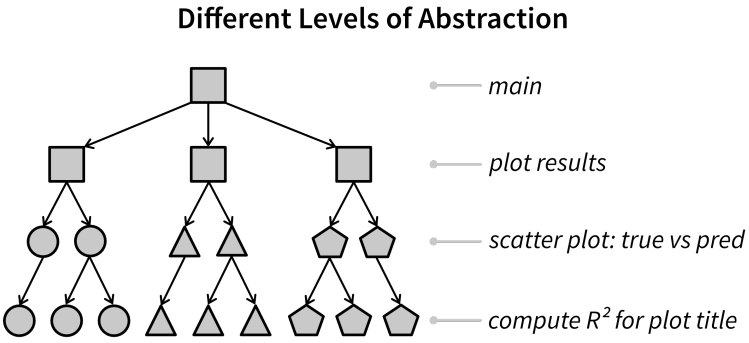


Figure 4.2: Complicated systems in software design can usually be represented as hierarchies that show different levels of abstraction, e.g., in this case for plotting the results of a predictive model, i.e., creating a scatter plot that shows the true vs. predicted values together with the R^2 value that indicates the overall performance of the model.

Elements of Software Systems

Before we continue, lets take a quick look at what the fundamental building blocks of our software systems are. To keep things simple, we distinguish between:

- **Variables:** Used to store data values, either primitive (like integers or strings, like we saw in Chapter 2) or composite, like lists and dictionaries, which can grow arbitrarily complex in some languages (e.g., a list can contain multiple dictionaries that themselves contain lists etc.).

```
# 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,
}
```

- **Functions:** Used to calculate something and/or perform an action, usually given some input arguments. We distinguish between pure and impure functions. A **pure function** is similar to a mathematical function $f : x \rightarrow y$ that computes and returns y given some x. **Impure functions** have so-called "side effects", i.e., they perform some action that has effects that are visible outside of the function itself, e.g., because they create or modify a file. They also don't necessarily have a return value.

```
def add(a, b):
    # pure: a simple calculation
    return a + b

def create_plot(x, y):
    # impure: create and save a plot
    plt.plot(x, y)
    plt.savefig("my_figure.png")
```

- **Objects:** They are basically a combination of variables and functions, i.e., specific constructs that store data values and have functions that usually access and modify these values. This comes in handy if you want to group multiple variables in one place because they logically belong together (e.g., the parameters used to configure a simulation model) and you want to use them store the results from functions.

```
class MyModel:

  def __init__(self, param1, param2):
      # initialize values from input arguments
      self.param1 = param1
      self.param2 = param2
      # create additional variables
```

```
        self.results = []

    def run_simulation():
        # do something
        self.result = [1, 2, 3]
```

System Boundaries

To avoid creating a complex mess, it is very important that we are clear on our (sub)system's boundaries, i.e., what is in and what is out of our code's scope. Adhering to clear boundaries makes it very easy to change our code. Unfortunately, though, establishing clear boundaries is easier said than done.

Let's start with a very simple example to illustrate the concept of scope:

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

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

Our code of interest (i.e., the subsystem we're looking at) is the function `n_times_x`. Inside the scope of this function with have the variables `x`, `n`, `i`, and `result`, i.e., when we can refer to them by name and when we execute the code it is clear what we're referring to. Outside of the scope of the function, where it is called in the program's `__main__` function, we have the variables `new_x`, `new_n`, and `my_result`. If we tried to access any of the internal variables from `n_times_x` here, we would get an error, since these variables are hidden inside the scope of `n_times_x`.

> **Note**
>
> While under some circumstances, depending on how you structure your code, the "inside code" could access the "outside code"'s variables, it is best to avoid this, i.e., try to have a clean separation between what is going on inside and outside of your code. Ideally, your code should only be concerned about what is going on inside your code and not depend on anything that exists outside of it. Therefore, if it needs access to any variables, you should just pass them as input arguments explicitly to make it clear what values your code uses.

This brings us to the boundaries, i.e., where "inside" and "outside" code meet. In software development, the boundary of your "inside code" is also referred to as its **interface**. This defines how you interact

with the code and it can be seen as a contract with the outside world for how to use your code. For a pure function, like in our example, the interface is the function's signature, i.e.,

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

As long as we keep this interface the same, i.e., the function still calculates the same thing, we're free to change whatever we want inside the function itself. For example, we could change the ridiculously inefficient implementation to use a proper multiplication:

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

And we don't have to tell anyone about this change, since we kept the interface of the function the same and none of the outside code was aware or depended upon what was going on inside our function. This is the beauty of clear boundaries.

On the other hand, if you decide that you don't like your function's name, changing this means that everywhere else where your function is used you also need to change the name. When you use an IDE to code, it probably has a feature to refactor your code where you can indicate that you want to change the name and then it checks all the files in your project for where this function is called and then changes the name there as well. However, if you're writing a library that is used in multiple place outside your current project, it will be very difficult to identify all the places where this function is used and notify the respective people to change the name. In practice this requires a deprecation process where the old interface is slowly phased out. This is why it really pays of to define good interfaces that remain stable for a long time.

> Go deep
>
> Powerful code has narrow interfaces with a deep implementation, i.e., the code does meaningful computations without exposing too much of its internals. For example, a narrow and deep function would maybe have two input arguments but extend over ten lines of code to do a complex calculation. When you split your code up into reusable functions, make sure that you don't split it up so much that you end up with a function that takes six input arguments but then only computes something on one line. Instead, try to create meaningful units of code that help you to hide complicated logic behind a simple interface and thereby make your code easier to change.

Pure functions with clean boundaries help us to write easily understandable code without any surprises. They are also easy to test, since they do not depend on anything besides what is passed as their input arguments, so no matter how many times you call a pure function with the same inputs, you're always going to get the same output. Your code gets more complex with **impure functions**: by definition, they have side effects and therefore interact and rely on the outside world. For example, they could write results to a file (e.g., create a plot) or read values from a database. This also means that if you run the code twice, you might get different results because something in the outside world has changed, e.g., your code crashes because the results file already exist or the output is different

because someone has changed the values in the database. This can make it harder to predict how changes to your code or somewhere else will affect the system as a whole, for example, because you don't know who else is accessing the same resources that your code uses.

As a best practice, you should always try to encapsulate as much critical logic as possible in pure functions, as this makes your code easier to understand and test. For example:

```
def pure_function(inputs):
    # do something without side effects
    output = ...
    return output

def impure_function():
    # read from outside file/database
    data = ...
    # do the main calculations
    result = pure_function(data)
    # write to outside file/database
    result.to_csv("result.csv")
```

⚠ Call-By-Value vs. Call-By-Reference

Depending on the programming language that you use, input arguments that are passed to a function can either be passed as their literal values (i.e., like getting a copy of the contents of the variable) or they can be passed as a reference, i.e., the function gets the location where the values reside in memory and can access (and manipulate!) them there. In the spirit of clean boundaries, we want to code by the principle "what happens inside a function stays inside the function (except for the return values)". However, when we pass a value by reference, this can result in unintended side effects (i.e., increase complexity), where we change the variables that was given to us from outside the function; a change which is then also visible outside of our code. In Python this can happen with complex data types like lists and dictionaries:

```
def change_list(a_list):
    a_list[0] = 42

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

To be safe, you can create a copy of a variable before you modify it to make sure the original stays the same and thereby avoid these side effects that could otherwise result in confusing bugs.

```python
from copy import deepcopy

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

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

Extend your scope with objects

Sometimes, your code might depend on so many variables that you don't want to pass all of these to a function. Especially when all of these variables and functionality are logically related, they should be grouped together into a class. A class, or an object, with is one concrete instantiation of a class, is great to create some extra scope for your functions.

Since we want clean boundaries and ideally a narrow interface, it is important to distinguish between *public* and *private* attributes and methods of a class. Everything about a class that is public is part of its interface, i.e., the contract with the outside world, which means changing these later can cause extra work or issues elsewhere. Private attributes and methods are only for internal use and can therefore be changed more easily.

> **i** Public and Private in different programming languages
>
> Depending on the programming language that you use, there might be more access levels than *public* and *private*. For example, in Java there also exists *protected* and *package*. For our purposes it is sufficient to distinguish between what should be accessible internally (for your code only) and externally (part of the contract with the outside world).
> In Python, nothing is really private as it is assumed that the programmer knows what she is doing when she accesses what she wants. By convention, variables and functions that are prefixed with `_` or `__` are for internal use only and you should only use them at your own risk. Therefore, it also makes sense to prefix all attributes and methods that you don't want anyone else to mess with with underscores. While this does not provide any access guarantees, at least who ever uses these variables will be warned that they may change without notice.

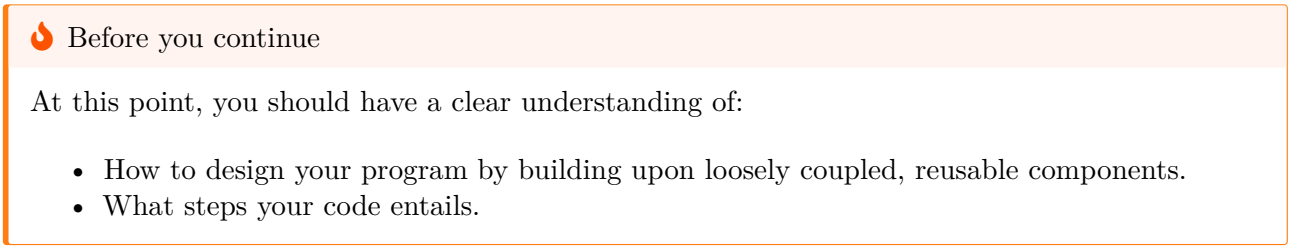
4.2 Draw Your How

Now it's time for you to draw your design.

But keep it simple—you don't have to overengineer your design to account for every possibility. If you want to build a one family home, design a one family home. Of course, it can't hurt to think ahead a little bit if you already know that some changes are likely to come ("How will the rooms change as the

kids move out?"), but there is no value in trying to plan ahead for everything ("What if the daughter wants to take over the house and transform it into an office for her 100+ people company?").

4.3 Make a Plan



5 Implementation

Now that you have a plan, it's finally time to get started with the implementation.

5.1 From Design to Code: Fill in the Blanks

Armed with your design from the last chapter, 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 X could result in the following draft:

> Order of functions
>
> Your script likely includes 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 general functions that only rely on external dependencies at the top (i.e., the ones that are at the lower levels of abstraction in your call hierarchy). 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.

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 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 …"*

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 hypotheses, 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. We recommend applying this 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.

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 solution can't outperform this baseline, something's off.

By starting with Minimum Viable Results, you can **test your code end-to-end** early on, see tangible progress, and **iteratively improve** from there.

Breaking Code into Components

Starting a new project often begins with all your code in a single script or notebook. 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 called (sub)modules. **Separating code into modules makes your project easier to navigate, test, and reuse.**

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 `preprocessing.py`, `models.py`, or `plot_results.py`. This modular approach naturally leads to a clean directory structure, which might look like this for a larger Python project:[1]

```
src/
  my-package/
      __init__.py
      main.py
      models/
```

[1] 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.

```
        __init__.py
        baseline_a.py
        baseline_b.py
        my_model.py
    utils/
        __init__.py
        preprocessing.py
        plot_results.py
```

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

```python
from models.my_model import MyModel
from utils import preprocessing

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 `utils/preprocessing.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 dependencies 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 others to install and use it just like any other external library.

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.
- **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 [8].

- **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 [1] 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.

5.2 Documentation & Comments: A Note to Your Future Self

While you write it, everything seems obvious. However, when revisiting your code a few months later (e.g., to address reviewer feedback), 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 parameters, 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, 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[2]

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.
- **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., `color_dict`), since if you decide to change the data type later, you either need to rename the variable everywhere or the name is now misleading.

5.3 Tests: Protect What You Love

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 ensure correctness initially, it becomes cumbersome to recreate these test cases later when the

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

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:

```
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"
```

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 dependencies.

> **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.

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, 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 components (e.g., single functions) to verify basic logic.
> - **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.
> - **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.

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 none of your tests caught the bug, write a new test to reproduce it and ensure this case is covered in the future.

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 that not only fixes bugs but also improves your understanding of the code and its behavior.

5.4 Make It Fast

> *Make it run, make it right, make it fast.*
> – Kent Beck (or rather this dad, Douglas Kent Beck[3])

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 never 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 quick way to find bottlenecks is to manually interrupt your code during a long run; if it always stops in the same place, that's likely 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.

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

> **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**. This "divide and conquer" approach is used in many algorithms, like the merge sort algorithm, and in big data frameworks like MapReduce.

Example: MapReduce

MapReduce [3] 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're training 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**: Transforms each element in a sequence.
- **filter**: Keeps elements that meet a condition.
- **reduce**: Aggregates elements recursively (e.g., summing values).

For example:

```
from functools import reduce

### Simplify this loop:
current_sum = 0
current_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:
        current_sum += new_i
        current_max = max(current_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 value)
current_sum = reduce(lambda acc, x: acc + x, new_i_filtered, 0)
current_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.

5.5 Refactoring: Make Change Easy

Refactoring is the process of **modifying existing code without altering its external behavior**. 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 variables, functions, or classes more meaningful and descriptive names.
- **Extracting Functions**: Breaking large functions into smaller, more focused ones (→ one function should do one thing).
- **Eliminating Duplication**: Consolidating repeated code into reusable functions.
- **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.
   - 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 a more general, reusable functions and classes.

Refactorings to simplify changes

> *For each desired change, make the change easy (warning: this may be hard), then make the easy change.*
> – Kent Beck[4]

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

- **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:
    ...
```

- **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():
    ...
```

- **Organize for Coherence**: 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, it signals a need to reorganize the code.** Clear ownership and reduced dependencies help teams work independently while keeping the system coherent through agreed upon interfaces.

Additional tips

- **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.

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

Before you continue

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

- How to transform your ideas into code.
- Some best practices to write code that is easy to understand and maintain.

6 From Research to Production

6.1 Components of Software Products

Graphical User Interface (GUI)

Databases

APIs

Events

Batch Jobs

Software Design Revisited

6.2 Delivery & Deployment

Continuous Integration

Containers in the Cloud

Containers like **Docker** capture the entire computing environment, making it portable and consistent across machines. A simple example `Dockerfile` for a Python project:

```
FROM python:3.9
WORKDIR /app
COPY requirements.txt requirements.txt
RUN pip install -r requirements.txt
COPY . .
CMD ["python", "main.py"]
```

Testing & Production Environments

Scalability

Observability

> 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

References

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

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

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

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

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

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

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

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