Introduction to Data Science

dr Maciej Świtała
Ewa Weychert

Class 2: Computer programming for data science

Class 2: What to Expect

- Overview of different programming languages
- Basic issues you may encounter in Python
- Basic errors in Python
- GitHub
- vnev - virtual environment
- Basic math behind data science and machine learning

Where Programming Fits in Data Science?

- **The Data Science Lifecycle:**
  - **Acquire:** Collect raw data from databases, APIs, sensors, or web scraping.
  - **Clean:** Handle missing values, remove duplicates, and transform data into usable formats.
  - **Analyze:** Apply statistical models, machine learning, and exploratory data analysis.
  - **Communicate:** Visualize results, build dashboards, and generate reports for stakeholders.
  - **Deploy:** Integrate models and insights into real-world systems or decision pipelines.
- **Programming as the Glue:**
  - Bridges raw data and actionable insights.
  - Enables interaction between statistics, algorithms, and domain knowledge.
  - Supports iterative experimentation and continuous improvement.

Where Programming Fits in Data Science?

- **Key Benefits of Programming in Data Science:**
  - **Automation:** Eliminate repetitive manual steps.
  - **Reproducibility:** Ensure analyses can be repeated and verified.
  - **Scalability:** Handle large datasets efficiently using code and computation.
  - **Auditability:** Maintain transparent, version-controlled workflows.
  - **Flexibility:** Adapt to new data sources, methods, or business needs quickly.

- **In essence:** Programming transforms data science from a collection of tools into a coherent, efficient process for generating reliable, data-driven insights.

Why Multiple Languages Matter (1/2)

- **Different Tasks Require Different Strengths:** Each stage of the data workflow benefits from specific languages.
  - **ETL (Extract, Transform, Load):** Python and SQL (Structured Query Language) excel in automating data ingestion, cleaning, and transformation pipelines.
  - **Modeling and Analysis:** R (a statistical computing language) and Python provide rich ecosystems for machine learning, visualization, and statistical inference.
  - **Deployment and Production:** Java, Scala, and Go (Golang) are well-suited for scalable, high-performance services that integrate models into real-time applications.

Why Multiple Languages Matter

- **Cross-Language Integration and Communication** Modern data science depends on seamless communication between tools and platforms.
  - **APIs (Application Programming Interfaces):** Enable integration between services written in different languages.
  - **Data Formats:** Parquet (columnar storage format) and Arrow (in-memory data format) support efficient cross-language data exchange.
  - **Microservices:** Independent components that communicate via APIs, allowing teams to mix languages (e.g., Python for analytics, Java for backend).
- **Career Agility and Collaboration:** Data scientists and engineers benefit from understanding multiple programming paradigms.
  - Adapt quickly to diverse tech stacks and project requirements.
  - Communicate effectively with teams using different languages.
  - Choose the best tool for the task—balancing speed, scalability, and maintainability.

Programming Languages in Data Science

Language	Primary Role in Data Science	Common Users / Communities
Python	General-purpose language for data analysis, machine learning, and automation. Rich ecosystem (NumPy, pandas, scikit-learn, TensorFlow).	Data scientists, ML engineers, researchers.
R	Statistical computing and visualization. Strong for exploratory data analysis and academic research.	Statisticians, data analysts, academics.
SQL (Structured Query Language)	Extracting and manipulating data from relational databases.	Data analysts, database administrators, data engineers.
Java	Robust, scalable language for production-grade pipelines and enterprise data systems.	Software engineers, backend developers, enterprise teams.
Julia	Designed for high-performance numerical computing and scientific research.	Computational scientists, researchers, quantitative analysts.

Tabela 1: *

Each language has unique strengths — mastering several builds flexibility across the entire data science

Python: Strengths in Data Science (1/2)

- **Extensive Ecosystem and Libraries:** Python offers a vast range of mature, open-source libraries for every stage of the data science workflow.
    - **NumPy:** Efficient numerical computations and array operations.
    - **pandas:** Data manipulation and analysis using intuitive tabular structures.
    - **scikit-learn:** Machine learning algorithms and model evaluation tools.
    - **PyTorch** and **TensorFlow:** Deep learning frameworks for research and production.
- **Versatile and General-Purpose:** Python is not limited to data analysis — it can automate workflows, connect databases, integrate APIs (Application Programming Interfaces), and orchestrate end-to-end pipelines.
    - Acts as a “glue language” connecting tools written in R, C++, or Java.
    - Simplifies experimentation with readable, high-level syntax.
- **Interactive and Productive Development Environment:** The Python ecosystem supports rapid experimentation and visualization.

Python: Essential Libraries

- **Numerical:** numpy, scipy
- **Data Handling:** pandas, polars
- **Machine Learning:** scikit-learn, pytorch, tensorflow
- **Visualization:** matplotlib, plotly, seaborn
- **Production / Deployment:** fastapi, pydantic

```
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
import pandas as pd

# Generate synthetic data
x = np.linspace(0, 10, 100)
y = np.sin(x) + np.random.randn(10
df = pd.DataFrame({'x': x, 'y': y}

# Visualization
sns.scatterplot(data=df, x='x', y=
plt.title("Noisy Sine Wave")
plt.xlabel("X-axis")
plt.ylabel("Y-axis")
plt.show()
```

R: Strengths in Data Science

- **Statistics-First Language:** Built for statistical modeling and data exploration from the ground up.
  - Rich base functions for hypothesis testing, regression, and inference.
  - Extensive CRAN (Comprehensive R Archive Network) packages for every analysis type.
- **The Tidyverse Ecosystem:** A cohesive set of packages (`dplyr`, `tidyr`, `readr`, `ggplot2`) that simplify data manipulation, cleaning, and visualization. Ideal for rapid EDA (Exploratory Data Analysis).
- **Reproducibility and Reporting:** `R`

```
# Load tidyverse
library(tidyverse)

# Create synthetic data
df <- tibble(
  x = seq(0, 10, length.out = 100)
  y = sin(x) + rnorm(100, sd = 0.2
)

# Visualization with ggplot2
ggplot(df, aes(x = x, y = y)) +
  geom_point(color = "steelblue")
  geom_smooth(method = "loess",
              se = FALSE, color =
  labs(title = "Noisy Sine Wave",
       x = "X-axis", y = "Y-axis")
```

R vs. Python: When to Choose Each

- **R — Best for Statistical Depth and Exploration:** Designed by statisticians for statistical modeling, visualization, and rapid data exploration.
  - Excellent for EDA (Exploratory Data Analysis) and visualization through `ggplot2` and the `tidyverse`.
  - Rich libraries for advanced statistics, bioinformatics, and time-series analysis.
  - Seamless integration with `R Markdown` and `Quarto` for reproducible, publication-ready reports.

- **Python — Best for Production and Scalability:** A general-purpose programming language that scales from research to deployment.
  - Strong in machine learning and deep learning via `scikit-learn`, `TensorFlow`, and `PyTorch`.
  - Easy integration with databases, web APIs, and cloud platforms (`FastAPI`, `Flask`).
  - Ideal for large-scale production systems, automation, and end-to-end data pipelines.

R vs. Python: When to Choose Each

- **Interoperability and Team Context:**
  - Tools like `reticulate` (R–Python interface) and `Arrow` (cross-language columnar data format) enable hybrid workflows.
  - Choose the language that fits your team's expertise and existing infrastructure — interoperability can fill the gaps.

Sources: RStudio (Posit) – *R for Data Science* (Wickham & Grolemund, 2017); Van Rossum, G. (1995) – *Python Tutorial*; KDnuggets (2024) – "R vs Python for Data Science: Which Should You Choose?"; Stack Overflow Developer Survey (2024).

Difference Between R and Python

Feature	**R**	**Python**
Introduction	Language and environment for statistical computing and graphics.	General-purpose programming language for data analysis and scientific computing.
Objective	Ideal for statistical analysis and visualization.	Suitable for application development, automation, and embedded systems.
Workability	Contains specialized packages for analytical tasks.	Supports matrix computation, optimization, and integration.
IDE	RStudio, RKward, R Commander.	Spyder, Jupyter, Eclipse+PyDev, Atom.
Libraries	`ggplot2`, `caret`, `dplyr`.	`pandas`, `NumPy`, `SciPy`.
Scope	Used for advanced statistical analysis.	Offers a streamlined approach to data science workflows.

Ecosystem in R and Python

Python Ecosystem:

- Large community for general-purpose data science.
- Rich data-centric packages: `pandas`, `NumPy`, `matplotlib`.
- Simplifies importing, analyzing, and visualizing data.

R Ecosystem:

- Strong in standard machine learning and statistical analysis.
- Packages for data mining and visualization: `tidyverse`, `ggplot2`, `caret`.
- Excellent for hypothesis testing and probabilistic modeling.

R vs. Python in Data Science Workflow

Feature	**R**	**Python**
Data Collection	Imports data from Excel, CSV, and text files.	Handles all data formats, including SQL tables and APIs.
Data Exploration	Optimized for statistical analysis of large datasets.	Uses `pandas` for data wrangling and EDA.
Data Modeling	Supports `tidyverse` and `caret`.	Uses `NumPy`, `SciPy`, `scikit-learn`, `TensorFlow`.
Data Visualization	`ggplot2` for complex visualizations.	`matplotlib`, `pandas`, `seaborn`.

Statistical Analysis and Machine Learning

Capability	**R**	**Python**
Basic Statistics	Built-in functions (mean, median, etc.)	NumPy (mean, median, etc.)
Linear Regression	`lm()` function	`statsmodels.OLS()`
Generalized Linear Models	`glm()` function	`statsmodels.GLM()`
Time Series	`forecast` package	`statsmodels.tsa`
ANOVA, t-tests	`aov()`, `t.test()`	`scipy.stats`
Hypothesis Tests	`wilcox.test()`	`scipy.stats.mannwhitneyu()`
PCA	`princomp()`	`scikit-learn.PCA()`
Clustering	`kmeans()`, `hclust()`	`scikit-learn.KMeans()`
Decision Trees	`rpart()`	`DecisionTreeClassifier()`
Random Forest	`randomForest()`	`RandomForestClassifier()`

Advantages of R and Python

R Programming	**Python Programming**
Supports large datasets for statistical analysis.	General-purpose programming for diverse tasks.
Favored by scholars and R&D professionals.	Widely used by developers and data engineers.
Strong libraries: **`tidyverse`**, **`ggplot2`**, **`caret`**, **`zoo`**.	Libraries: **`pandas`**, **`scikit-learn`**, **`TensorFlow`**, **`NumPy`**.
Integrated IDE: RStudio.	Integrated with Conda, Spyder, Jupyter.

Disadvantages of R and Python

R Programming	**Python Programming**
More difficult to learn; focused on statistical tasks.	Fewer built-in statistical libraries compared to R.
Slower for large-scale computation.	Less specialized for advanced statistics.
Memory management can be inefficient for very large datasets.	Visualization not as refined as `ggplot2`.

SQL: The Lingua Franca of Data

- **Structured Query Language (SQL):** A domain-specific language used to manage and query relational databases.
  - The Lingua Franca - term that refers to a bridge language or common language used among people with different native languages to enable communication. Historically, it referred to trade languages like Mediterranean Lingua Franca or modern English in global business.
  - Core operations: `SELECT`, `INSERT`, `UPDATE`, `DELETE`.
  - Used in virtually every data system — from small SQLite files to massive cloud warehouses.
- **Query at Scale:** SQL enables direct computation inside data warehouses like **Google BigQuery**, **Snowflake**, and **Amazon Redshift**.
  - Push computation to where data lives — avoid downloading gigabytes locally.
  - Combine performance and security through managed infrastructure.
- **Analytical Power:** SQL remains essential for transforming, aggregating, and summarizing data efficiently.
  - Master key constructs: `JOINs`, `GROUP BY`, `HAVING`, `CASE WHEN`

SQL: Example

```
SELECT user_id,
       COUNT(*) AS n_sessions,
       AVG(session_len) AS avg_len,
       PERCENTILE_CONT(0.9) WITHIN GROUP (ORDER BY session_len) AS p90
FROM sessions
WHERE session_date >= CURRENT_DATE - INTERVAL '30' DAY
GROUP BY user_id
HAVING COUNT(*) >= 3
ORDER BY p90 DESC
LIMIT 20;
```

SQL JOINs: Overview

- **Goal:** Combine rows across tables using key relationships.
- **Common JOINs:** *INNER*, *LEFT (OUTER)*, *RIGHT (OUTER)*, *FULL (OUTER)*, *CROSS*, *SELF*.
- **Join condition:** `ON` (general predicate) or `USING(col)` (shorthand when column names match).
- **NULLs:** Outer joins preserve unmatched rows with `NULL`s on the other side.

Schema: Customers & Orders

Tables and Keys

- `customers`(<u>customer_id</u>, name, city)
- `orders`(<u>order_id</u>, *customer_id* FK $\rightarrow$ `customers.customer_id`, order_total)

ER Sketch

```
customers                  orders
----------                 -----------
customer_id (PK) <---- FK  customer_id
name                       order_id (PK)
city                       order_total
```

Sample Rows

customers

customer_id	name	city
1	Ada	London
2	Linus	Helsinki
3	Grace	New York
4	Margaret	London

orders

order_id	customer_id	order_total
10	1	120.00
11	1	80.00
12	3	60.00
13	5	50.00

Note: `orders.customer_id=5` has no matching customer (orphans). Customer #2 and #4 have no orders.

INNER JOIN

Keeps only matching rows (intersection).

SQL Query

```
SELECT c.customer_id, c.name,
       o.order_id, o.order_total
FROM customers AS c
INNER JOIN orders AS o
  ON o.customer_id = c.customer_id
ORDER BY c.customer_id, o.order_id;
```

Result Set

customer_id	**name**	**order_id**	**order_total**
1	Ada	10	120.00
1	Ada	11	80.00
3	Grace	12	60.00

Only customers with matching `customer_id` in both tables appear.

LEFT (OUTER) JOIN

Keeps all rows from the left table and fills unmatched rows with `NULL`.

SQL Query

```
SELECT c.customer_id, c.name,
       o.order_id, o.order_total
FROM customers AS c
LEFT JOIN orders AS o
  ON o.customer_id = c.customer_id
ORDER BY c.customer_id, o.order_id;
```

Result Set

customer_id	name	order_id	order_total
1	Ada	10	120.00
1	Ada	11	80.00
2	Linus	NULL	NULL
3	Grace	12	60.00
4	Margaret	NULL	NULL

Note: Order with `customer_id = 5` is excluded (no match in left table).

RIGHT JOIN

Keeps all rows from the right table (orders) and fills unmatched left rows with `NULL`.

SQL Query

```
SELECT c.customer_id, c.name,
       o.order_id, o.order_total
FROM customers AS c
RIGHT JOIN orders AS o
  ON o.customer_id = c.customer_id
ORDER BY o.order_id;
```

Result Set

customer_id	name	order_id	order_total
1	Ada	10	120.00
1	Ada	11	80.00
3	Grace	12	60.00
NULL	NULL	13	50.00

Note: Order #13 (customer_id = 5) appears even though no matching customer exists.

FULL OUTER JOIN

Combines LEFT and RIGHT JOINs — keeps all rows from both tables.

SQL Query

```
SELECT c.customer_id, c.name,
       o.order_id, o.order_total
FROM customers AS c
FULL OUTER JOIN orders AS o
  ON o.customer_id = c.customer_id
ORDER BY c.customer_id, o.order_id;
```

Result Set

customer_id	name	order_id	order_total
1	Ada	10	120.00
1	Ada	11	80.00
2	Linus	NULL	NULL
3	Grace	12	60.00
4	Margaret	NULL	NULL
NULL	NULL	13	50.00

Shows all customers and all orders — unmatched records from either table appear with NULLs.

Julia: Modern Language for Scientific Computing

- **Designed for numerical performance** with syntax as easy as Python but speed close to C.
  - Compiles to efficient machine code using LLVM.
  - Excellent for linear algebra, simulations, and numerical optimization.
- **Multiple dispatch:** functions automatically choose the best implementation based on argument types — a powerful feature for scientific modeling.
- **Growing ML and data ecosystem:**
  - `Flux.jl`, `MLJ.jl`, `DataFrames.jl`, and `Plots.jl`.
  - Actively used in research, numerical analysis, and computational physics.
- **Interoperability:** Can call C, Python, or R directly — useful for hybrid pipelines.

Scala (Apache Spark Ecosystem)

- **JVM-based, strongly typed language** combining functional and object-oriented paradigms.
- **Primary language for Apache Spark:**
    - Used to build distributed data processing and ETL pipelines.
    - Ideal for working with petabyte-scale data and real-time streaming.
- **When to use:**
    - Enterprise environments already using the Java Virtual Machine (JVM).
    - Scenarios where low latency, parallel computation, and data scalability are essential.
- Integrates well with Java libraries and Spark SQL / DataFrames APIs.

Java: Enterprise Reliability and Scalability

- **Mature, platform-independent language** widely used in enterprise data systems.
- **Applications in Data Science:**
  - Backend services, stream processing (`Kafka`, `Flink`).
  - High-throughput data ingestion and real-time model serving.
- **Machine Learning Integration:**
  - Many ML frameworks provide Java APIs — e.g., `TensorFlow Java`, `DL4J`, `H2O.ai`.
  - Useful for deploying trained models as scalable REST services.
- **Strengths:** reliability, portability, multithreading, and strong community support.

C / C++: The Power Behind the Scenes

- **Low-level control and high performance:** Ideal for compute-heavy applications, such as simulations, GPU kernels, and custom ML operations.
- **Under the hood:** Many Python and R libraries (e.g., `NumPy`, `TensorFlow`, `caret`) rely on C/C++ backends.
- **When to use:**
  - Building performance-critical components or embedded AI systems.
  - Writing custom operations for deep learning frameworks.
- **Domains:** HPC (High Performance Computing), robotics, edge devices, and real-time analytics.

MATLAB / Octave: Numerical and Academic Roots

- **MATLAB:** Proprietary environment widely used in academia, research, and engineering.
- **GNU Octave:** Open-source alternative largely compatible with MATLAB syntax.
- **Strengths:**
  - Extensive toolboxes for signal processing, control systems, image analysis, and simulations.
  - Powerful for prototyping algorithms before converting to production code.
- **Limitations:**
  - Limited scalability for large distributed systems.
  - Typically ported to Python or C++ for production use.

JavaScript: Visualization and Web Applications

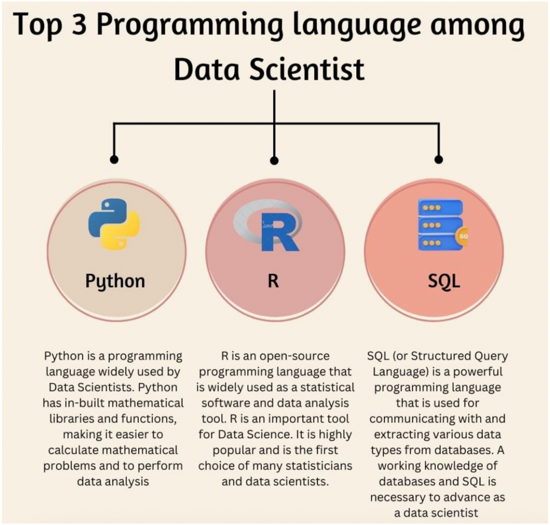
- **Front-end language of the web** — used to build interactive analytics dashboards and visualizations.
- **Visualization Libraries:**
  - `D3.js`, `Plotly.js`, `Chart.js`, `Vega`, and `Observable`.
- **Data Science Integration:**
  - Acts as the front-end layer for Python or R-based machine learning services via APIs.
  - Supports real-time visualization of predictions and metrics in the browser.
- **Advanced Use:**
  - `WebAssembly (Wasm)` and `TensorFlow.js` enable ML directly in the browser.
  - Excellent for deploying lightweight AI and data-driven web apps.

Choosing the Right Tool for the Job

- **Start with Team-Native Language and Infrastructure:** Build on the tools your team already knows and supports.
  - R for analytics-heavy teams; Python for end-to-end data pipelines.
  - Scala/Java when working within JVM ecosystems (e.g., Spark, Kafka).
  - Leverage existing deployment infrastructure (cloud, CI/CD, monitoring).

- **Optimize for Maintainability and Observability:** Data projects must be sustainable and transparent.
  - Write clear, version-controlled code with reproducible results.
  - Build automated testing, data validation, and logging into pipelines.
  - Prioritize readable workflows over clever one-liners.

- **Interoperability and Integration:** No single tool does everything — connect systems effectively.
  - Use standardized data formats such as `Parquet`, `Arrow`, and `Avro`.
  - Integrate components through **APIs**, **message queues** (Kafka, RabbitMQ), or cloud storage.
  - Combine strengths: prototype in R/Python, scale in Spark or SQL.



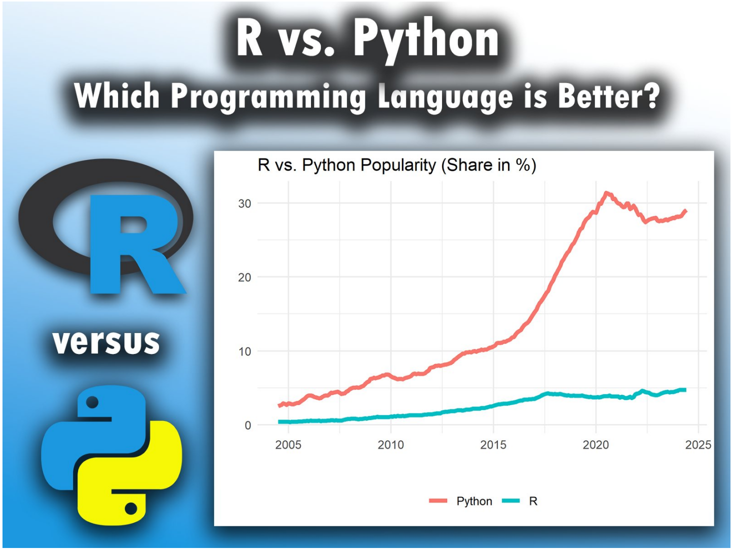
Top 3 Programming language among Data Scientist

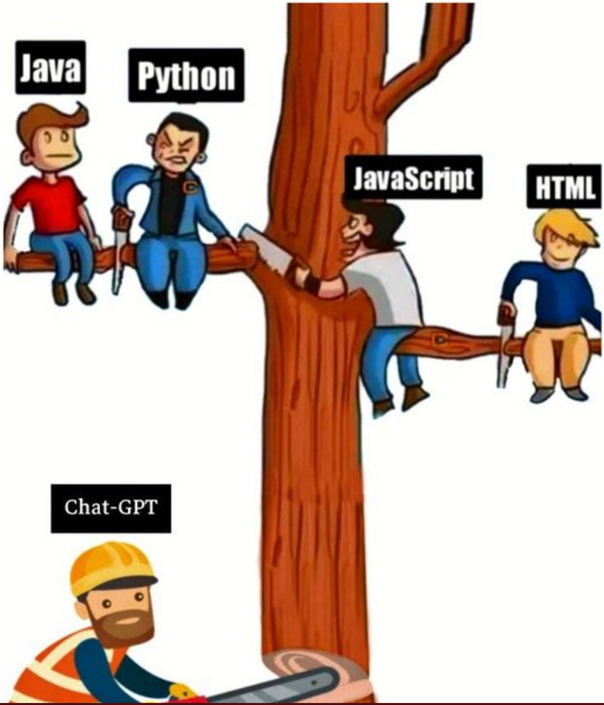
Python
R
SQL

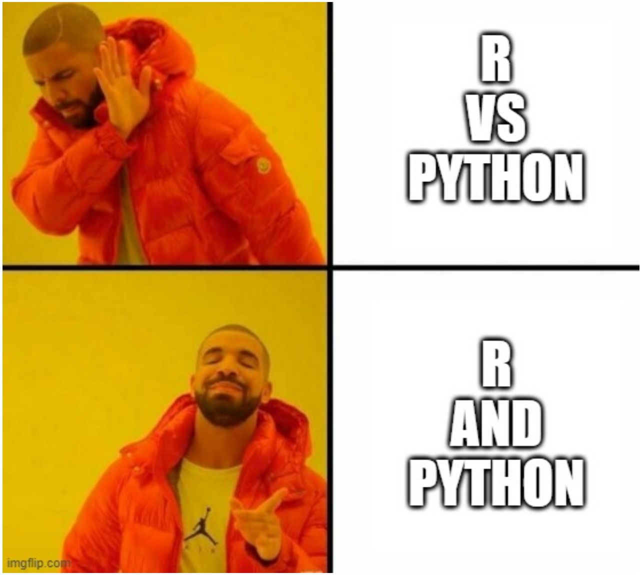
Python is a programming language widely used by Data Scientists. Python has in-built mathematical libraries and functions, making it easier to calculate mathematical problems and to perform data analysis

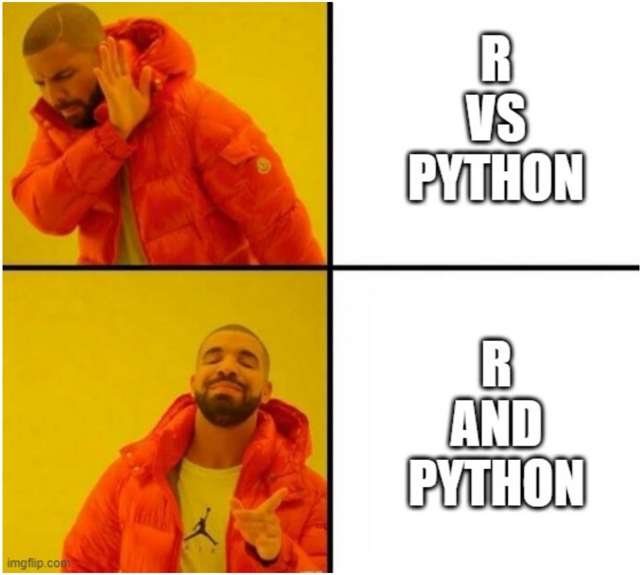
R is an open-source programming language that is widely used as a statistical software and data analysis tool. R is an important tool for Data Science. It is highly popular and is the first choice of many statisticians and data scientists.

SQL (or Structured Query Language) is a powerful programming language that is used for communicating with and extracting various data types from databases. A working knowledge of databases and SQL is necessary to advance as a data scientist

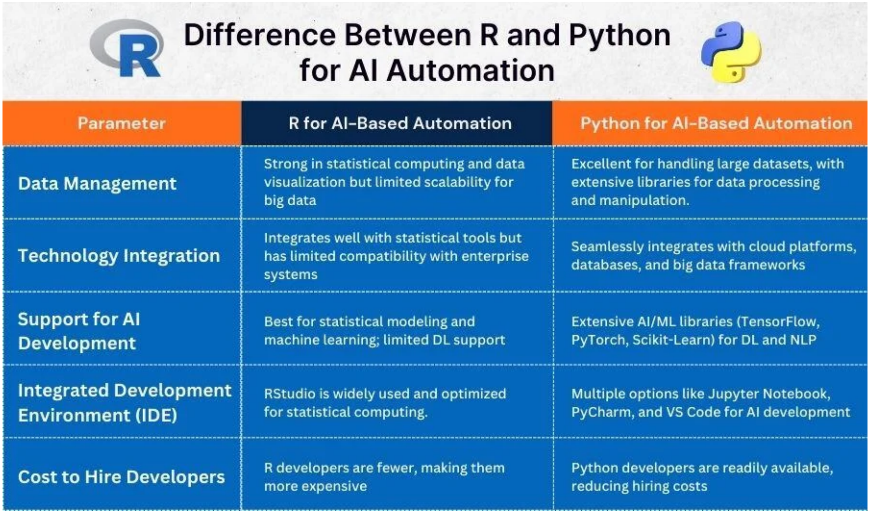

R vs. Python
Which Programming Language is Better?
versus
R vs. Python Popularity (Share in %)
30
20
10
0
2005
2010
2015
2020
2025
Python
R


Java
Python
JavaScript
HTML
Chat-GPT


R
VS
PYTHON
R
AND
PYTHON
imgflip.com

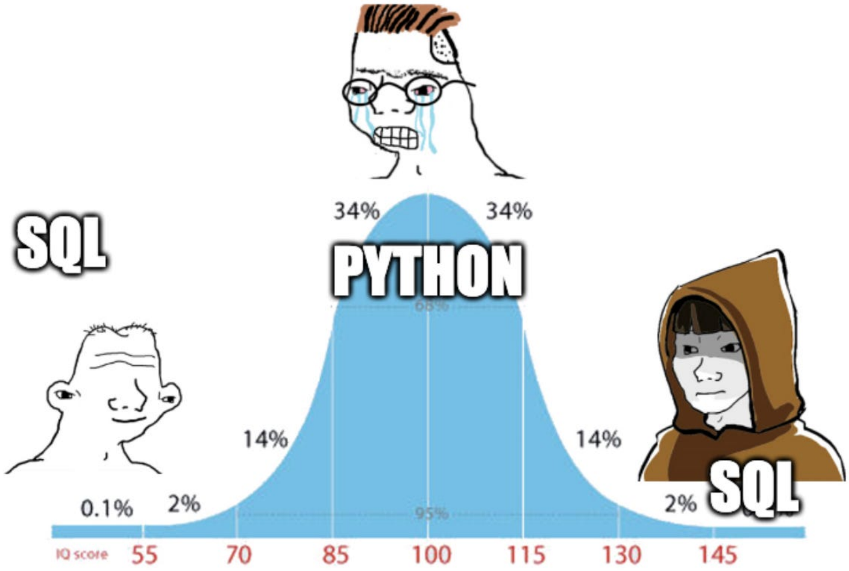

R
VS
PYTHON
R
AND
PYTHON
imgflip.com

A1
=1+2
A
B
C
D
1
3
2
3
4
data siens



Difference Between R and Python for AI Automation

Parameter	R for AI-Based Automation	Python for AI-Based Automation
Data Management	Strong in statistical computing and data visualization but limited scalability for big data	Excellent for handling large datasets, with extensive libraries for data processing and manipulation.
Technology Integration	Integrates well with statistical tools but has limited compatibility with enterprise systems	Seamlessly integrates with cloud platforms, databases, and big data frameworks
Support for AI Development	Best for statistical modeling and machine learning; limited DL support	Extensive AI/ML libraries (TensorFlow, PyTorch, Scikit-Learn) for DL and NLP
Integrated Development Environment (IDE)	RStudio is widely used and optimized for statistical computing.	Multiple options like Jupyter Notebook, PyCharm, and VS Code for AI development
Cost to Hire Developers	R developers are fewer, making them more expensive	Python developers are readily available, reducing hiring costs


SQL
PYTHON
SQL
34%
34%
68%
14%
14%
0.1%
2%
2%
95%
IQ score
55
70
85
100
115
130
145

Why GitHub for Data Science?

- **Version Control & Collaboration:** Seamlessly manage changes, track history, and work together using Git.
- **Reproducibility & Transparency:** Keep your code, data, and experiment configurations synchronized and traceable.
- **Open Science & Sharing:** Showcase your work publicly, contribute to others' projects, and build your portfolio.
- **Automation & CI/CD:** Integrate testing, deployment, and data pipelines directly into your repositories.

Introducing Fork: A Friendly Git GUI

What is Fork?

A fast, intuitive Git client for macOS and Windows. It offers GUI-based operations while exposing powerful Git workflows.

Key Features

- Commits visually.
- Built-in merge-conflict helper and resolver.
- Commit history, blame, and repository browsing at any commit.

Why use it in data science / software projects?

- Makes advanced Git workflows (rebasing, conflict resolution) more accessible.
- Visual diff tools help when dealing with code, notebooks, or image output.
- Lowers the barrier for newcomers to use Git aggressively and confidently.

Forking in Data Science Teams

- **Experiment Safely:** Forking allows each data scientist to prototype new models, feature engineering strategies, or visualizations independently before merging back into production.
- **Reproducibility:** Forked repositories capture the exact state of code, notebooks, and configurations used in experiments.
- **Collaboration:** Combine forks with GitHub features:
    - **Pull Requests:** peer review and discussion.
    - **Issues:** track bugs or feature ideas.
    - **Actions:** automate testing or notebook execution on each PR.
- **Integration:** Fork-based workflows fit easily into MLOps systems — CI/CD pipelines can automatically test and validate code before merging.

In Essence: Why Forking Matters

- Promotes open, modular, and collaborative development.
- Encourages experimentation without risk to production code.
- Enables versioned, peer-reviewed improvements to shared data projects.
- Fosters a culture of transparency, contribution, and reproducible science.
- Forks are the backbone of open-source collaboration — a bridge between innovation and stability in data-driven research.

What is a Virtual Environment?

- **Definition:** A **virtual environment** is an isolated Python workspace that includes its own interpreter, dependencies, and libraries — separate from the system Python.
- **Purpose:**
  - Prevents dependency conflicts between projects.
  - Ensures **reproducibility** — the same code runs reliably across machines.
  - Keeps your system Python clean and stable.
- **Analogy:** Think of it as a "container" for each project — each can have different versions of `numpy`, `pandas`, or other libraries without interfering with others.
- **Common Tools:**
  - `venv` — built-in lightweight tool for environment creation.
  - `virtualenv` — legacy but flexible environment manager.
  - `conda` — full-featured environment + package manager (Python/R).
- **In Data Science:** Virtual environments are essential for consistent experiments, deployment, and collaboration — ensuring that results are reproducible from notebook to production.

Create & Activate a Virtual Environment

```
# create
python3 -m venv .venv

# activate (Linux/macOS)
source .venv/bin/activate

# activate (Windows PowerShell)
.\.venv\Scripts\Activate.ps1

# verify
python -V
which python
pip list
```

Locking dependencies

- `pip freeze > requirements.txt`
- Recreate: `pip install -r requirements.txt`
- Optional: `pip-tools` (`pip-compile`, `pip-sync`).

```
pip install --upgrade pip pip-tools
pip-compile pyproject.toml
pip-sync requirements.txt
```

Paths in Python: Absolute vs. Relative

- **Absolute Path:** Specifies the full path from the root of the file system.
  - Example (Linux/macOS): `/home/user/project/data/input.csv`
  - Example (Windows): `C:\Users\user\project\data\input.csv`
  - Always points to the same location, regardless of the current working directory.
- **Relative Path:** Defined in relation to the current working directory (CWD).
  - Example: `data/input.csv` or `../shared/utils.py`
  - Easier to share across machines — avoids user-specific paths.
- **Best Practice:**
  - Prefer paths relative to your **project root** or config file for portability.
  - Use Python's `pathlib` (introduced in Python 3.4) for cleaner, OS-independent code.
- **Example using `pathlib`:**

  `base = Path(`$_{file})$.parentcurrentscriptdirectorydata$_{f}$ile=base/"data"/"input.csv"print(data$_{f}$ile

- **In Data Science:** Path handling ensures that notebooks, scripts, and

os module basics

```
import os
print(os.getcwd())
os.makedirs("data/raw", exist_ok=True)
print(os.path.exists("data/raw"))
print(os.listdir("data"))
```

The `os` Module: Interacting with the Operating System

Purpose: The `os` module provides a portable way to interact with the file system, environment variables, and system-level operations directly from Python code.

Function	**Description**	**Example Output / Use**
`os.getcwd()`	Get current working directory	`'/home/user/project'`
`os.chdir(path)`	Change working directory	`os.chdir('data/')`
`os.listdir(path)`	List files in a directory	`['data.csv', 'model.pkl'`
`os.path.exists(path)`	Check if path exists	`True / False`
`os.makedirs(path)`	Create directories recursively	`os.makedirs('out/logs',`
`os.remove(path)`	Delete a file	`os.remove('temp.txt')`
`os.rename(src, dst)`	Rename or move file	`os.rename('a.csv', 'b.cs`
`os.environ`	Access environment variables	`os.environ['PATH']`
`os.system(cmd)`	Run a shell command	`os.system('ls -l')`

Why It Matters in Data Science:

- Automates data ingestion, cleaning, and file management tasks.
- Enables scripts to run seamlessly across operating systems.
- Integrates with environment variables for secure API keys and credentials.
- Supports reproducible pipelines — consistent folder and file handling.

Common Python Mistakes

Mutable Default Arguments

Problem: Default arguments in Python are evaluated *once at function definition time*, not each time the function is called. Mutable defaults (like lists or dicts) persist across calls.

```
# Bad
def append_item(x, items=[]):
    items.append(x)
    return items

print(append_item(1))  # [1]
print(append_item(2))  # [1, 2]  <-- unexpected shared list
```

Solution: Use `None` as the default and create a new list inside the function.

```
# Good
def append_item(x, items=None):
    items = [] if items is None else items
    items.append(x)
    return items
```

Tip: Always avoid mutable defaults (e.g., `[]`, `{}`, `set()`).

Off-by-One Errors

Problem: Loop boundaries and slicing indices are **exclusive at the end**, which often causes subtle logic errors.

```
for i in range(10):  # 0..9
    pass

s = "abcdef"
print(s[0:3])  # 'abc' (excludes index 3)
```

Common pitfalls:

- Miscounting iterations in `range()` loops.
- Using `<=` instead of `<` in while loops.
- Forgetting that slicing excludes the upper bound.

Best Practice: Use clear names (`end`, `limit`) and test edge cases explicitly.

Truthiness, Identity, and Equality

Problem: `is` tests **object identity**, while `==` tests **value equality**. Beginners often confuse them.

```
x = 256
print(x is 256)       # True (small ints cached by CPython)
print([1] == [1])     # True (same contents)
print([1] is [1])     # False (different objects)
```

Guidelines:

- Use `is` only for `None` or singletons (`True`, `False`).
- Use `==` for comparing values.
- Remember that `bool`, `int`, and `float` interact in non-obvious ways.

Shallow vs. Deep Copy

Problem: Copying containers with nested structures often results in shared references.

```
import copy
x = [[1],[2]]
shallow = x.copy()             # shallow copy
deep = copy.deepcopy(x)        # deep copy
x[0].append(9)
print(shallow)  # [[1, 9], [2]]
print(deep)     # [[1], [2]]
```

Explanation: `list.copy()` only copies the outer list; inner elements still reference the originals.

Best Practice: Use `copy.deepcopy()` for truly independent nested structures.

Floating Point Surprises

Problem: Decimal numbers cannot always be represented exactly in binary.

```
print(0.1 + 0.2)        # 0.30000000000000004
```

Solution: Use the `decimal` module for precise arithmetic.

```
from decimal import Decimal
print(Decimal('0.1') + Decimal('0.2'))  # 0.3
```

Why it matters:

- Small rounding errors can accumulate in financial or statistical calculations.
- Always format outputs when printing floating-point values.

Shadowing Built-ins

Problem: Redefining names that belong to Python's built-in functions prevents you from using them later.

```
list = [1, 2, 3]   # don't!
# later: list('abc')  # TypeError
```

Avoid names such as: `list`, `dict`, `set`, `id`, `type`, `sum`, `max`, `min`, `input`, `file`.

Best Practice: Use descriptive, domain-specific names (`data_list`, `student_dict`) instead.

Late Binding in Closures

Problem: Lambdas and inner functions capture variables by reference, not by value.

```
# Incorrect: all lambdas share the same i
funcs = [lambda: i*i for i in range(3)]
print([f() for f in funcs])  # [4, 4, 4]
```

Solution: Bind values at definition time using default arguments.

```
funcs = [lambda i=i: i*i for i in range(3)]
print([f() for f in funcs])  # [0, 1, 4]
```

Tip: Useful when defining callbacks or deferred computations in loops.

Pandas Chained Assignment

Problem: Chained indexing (e.g. `df[df.x > 0]['y'] = 0`) may operate on a copy, not the original DataFrame.

```
# Bad (may modify a temporary copy)
df[df.x > 0]['y'] = 0

# Good: Use .loc for explicit selection
mask = df.x > 0
df.loc[mask, 'y'] = 0
```

Explanation: Pandas may return a view or a copy depending on context — causing "SettingWithCopyWarning".

Best Practice: Always use `.loc[]` for assignment, and verify changes with `.head()` or `.equals()`.

Variable Scope and the Global Keyword

Problem: Variables defined inside a function are local by default. Attempting to modify a global variable without declaring it causes confusion or errors.

```
count = 0
def increment():
    count += 1  # UnboundLocalError!
    return count
```

Fix: Declare explicitly when using a global variable.

```
count = 0
def increment():
    global count
    count += 1
    return count
```

Tip: Avoid using globals for mutable state; prefer returning values or using classes.

Incorrect Use of `try/except`

Problem: Catching all exceptions hides real bugs and makes debugging difficult.

```
try:
    result = 1 / 0
except:
    print("Something went wrong")  # too broad
```

Fix: Catch only specific exceptions.

```
try:
    result = 1 / 0
except ZeroDivisionError:
    print("Cannot divide by zero.")
```

Tip: Never use a bare `except:` — always specify the expected error type.

Misusing List Comprehensions

Problem: List comprehensions are for creating lists, not for side effects.

```
# Bad: Using for side effects only
[print(x) for x in range(3)]
```

Fix: Use a regular for-loop for side effects, and comprehensions only when collecting results.

```
for x in range(3):
    print(x)
# Good use
squares = [x**2 for x in range(3)]
```

Tip: Keep list comprehensions readable; avoid nesting more than two levels.

Forgetting to Close Files

Problem: Unclosed file handles can lead to memory leaks or locked files.

```
# Bad
f = open("data.txt")
data = f.read()
# forgot: f.close()
```

Fix: Use context managers (`with`) to handle cleanup automatically.

```
with open("data.txt") as f:
    data = f.read()
# file closed automatically
```

Tip: Use `with` for any resource that needs cleanup: files, sockets, DB connections.

Misusing Boolean Operators

Problem: `and` / `or` return the last evaluated operand, not necessarily a boolean.

```
a = 0
b = 5
print(a or b)   # 5
print(a and b)  # 0
```

Explanation: `and` / `or` use short-circuit evaluation and return the actual operand.
Best Practice: Use explicit comparisons (`==`, `!=`, `>`) when testing values. For pure booleans, wrap expressions with `bool()` for clarity.

Summary: Common Python Pitfalls

Category	**Issue**	**Recommended Fix**
Functions	Mutable default args	Use `None` as default
Loops	Off-by-one errors	Test boundary conditions
Comparisons	`is` vs. `==`	Use `==` for equality
Copies	Shallow vs deep	Use `copy.deepcopy()`
Numbers	Floating precision	Use `Decimal` or rounding
Scope	Global variable misuse	Pass or return values
Classes	Shared mutable attributes	Define in `__init__()`
Files	Unclosed handles	Use `with open(...)`
Exceptions	Bare `except:`	Catch specific errors
Pandas	Chained assignment	Use `.loc[]` safely

Remember: Most bugs stem from misunderstanding scope, mutability, and implicit behavior.

Common Python Errors (Exceptions)

SyntaxError / IndentationError

Typical causes:

- Missing colons, mismatched braces/parentheses.
- Mixing tabs and spaces; inconsistent indentation levels.

Fixes & prevention: Use an auto-formatter (Black), enable editor visible whitespace, add a linter (Flake8/Ruff).

```
# Bad
if True
    print('Missing colon')   # SyntaxError

for i in range(3):
print(i)                     # IndentationError
```

Good:

```
if True:
    print('OK')

for i in range(3):
    print(i)
```

NameError / UnboundLocalError

Typical causes:

- Referencing a variable before assignment (`NameError`).
- Assigning to a name inside a function that also exists globally without `global`/`nonlocal` (`UnboundLocalError`).

Prefer: pass values in, return results out (pure functions).

```
def f():
    print(x)        # NameError: x not defined here
x = 10

def g():
    x = x + 1       # UnboundLocalError (x treated as local)
```

Safer patterns:

```
def f_ok(x):        # pass in
    print(x)

def g_ok(x):        # return out
    return x + 1
```

TypeError / AttributeError

Typical causes:

- Passing wrong types or arity (`TypeError`).
- Calling a method on `None` or wrong object (`AttributeError`).

Prevention: add type hints, validate inputs, handle `Optional`.

```
len(42)         # TypeError
x = None
x.append(1)     # AttributeError: 'NoneType' has no attribute 'append'
```

Guard pattern:

```
from typing import Optional, Sequence

def safe_len(xs: Optional[Sequence]) -> int:
    if xs is None:
        return 0
    return len(xs)
```

KeyError / IndexError

Typical causes:

- Accessing a missing dict key (`KeyError`).
- Accessing out-of-range list index (`IndexError`).

Safer access: `dict.get`, membership checks, bounds checks.

```
data = {'a': 1}
print(data['b'])               # KeyError
nums = [1, 2, 3]
print(nums[5])                 # IndexError
```

Safer:

```
value = data.get('b', 0)
if 0 <= 2 < len(nums):
    print(nums[2])
```

ValueError / AssertionError

Typical causes:

- Correct type, but invalid value (`ValueError`).
- Broken assumptions (`AssertionError`) during development/tests.

Validate early, fail fast:

```
int('abc')                # ValueError
import math
math.sqrt(-1)             # ValueError

def pct(x):
    if not 0 <= x <= 1:
        raise ValueError("x must be in [0,1]")
    return x
```

Assertions:

```
assert 2 + 2 == 4, "Invariant failed"
```

(Use assertions for internal invariants; not for user input validation.)

ImportError / ModuleNotFoundError

Typical causes:

- Package not installed, wrong env/venv active.
- Misspelled names, wrong import path.

Fixes: activate correct venv, pin versions, verify module symbols.

```
import non_existent_module     # ModuleNotFoundError
from math import squaroot      # ImportError (no such name)
```

Good hygiene:

```
# Activate venv, then:
pip install -r requirements.txt
python -c "import pkg; print(pkg.__version__)"
```

MemoryError / RecursionError

Typical causes:

- Loading gigantic data into memory; creating massive lists.
- Unbounded or deep recursion (`RecursionError`).

Prevention: stream/chunk data, use generators, prefer iteration.

```
# Huge allocation (avoid)
# x = [0] * 10**10   # MemoryError

def recurse():
    return recurse()

# recurse()          # RecursionError
```

Safer patterns:

```
def read_in_chunks(fp, size=1_000_000):
    while chunk := fp.read(size):
        yield chunk
```

FileNotFoundError / PermissionError

Typical causes:

- Wrong path, missing file/folders.
- Insufficient permissions (read/write/execute).

Best practices: use `pathlib`, check existence, create parents, `with` for I/O.

```
from pathlib import Path

p = Path("data/input.csv")
if not p.exists():
    print("Missing:", p)      # FileNotFoundError if you open() it

try:
    with open("/root/secret.txt", "w") as f:
        f.write("x")          # PermissionError
except PermissionError:
    print("No write permission")
```

Production Patterns: EAFP, Logging, Typed Interfaces

EAFP (Easier to Ask Forgiveness than Permission):

```
try:
    value = cache[key]
except KeyError:
    value = compute(key)
```

LBYL (Look Before You Leap):

```
if key in cache:
    value = cache[key]
else:
    value = compute(key)
```

Recommendations:

- Prefer EAFP with precise `except SomeError` blocks.
- Add logging and contextual info in exception paths.
- Use type hints + runtime validation for clearer contracts.

Summary of Common Python Errors

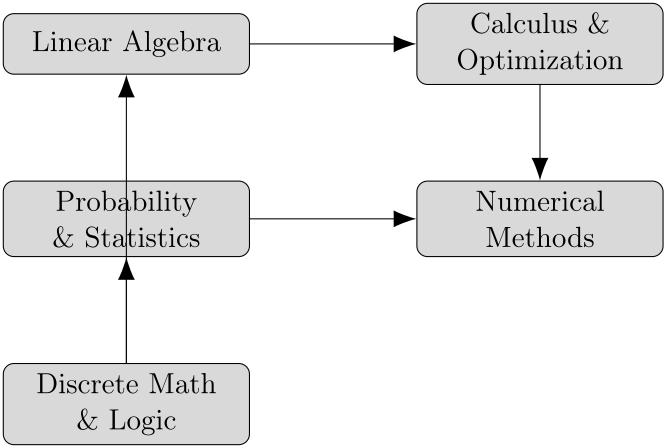
Error Type	**Cause**	**Prevention / Fix**
SyntaxError	Invalid syntax, missing colons	Use linter / Black formatter
IndentationError	Mixed tabs/spaces	Configure editor indentation
NameError	Undefined variable	Define or pass before use
UnboundLocalError	Local var shadows global	Use `global` or restructure
TypeError	Wrong argument type	Add input validation, type hints
AttributeError	Accessing missing attribute	Check for `None`, use `hasattr()`
KeyError	Missing dict key	Use `.get()` or `defaultdict`
IndexError	Index out of range	Check length before access
ValueError	Wrong value domain	Validate early, fail fast
AssertionError	Broken invariant	Fix logic, not input
ImportError	Wrong symbol import	Check package and spelling
ModuleNotFoundError	Missing dependency	`pip install -r requirements.txt`
MemoryError	Data too large	Stream or process in chunks
RecursionError	Infinite recursion	Add base case / iterate
FileNotFoundError	Missing file	Check path before open
PermissionError	Insufficient rights	Adjust permissions or path

Why Mathematics Matters in Data Science

Why Mathematics Matters in Data Science

- **Math is the language of models.**
- It helps you:
  - Understand algorithms, not just use them.
  - Diagnose overfitting, bias, and instability.
  - Build interpretable, explainable models.
  - Create your own custom ML or AI methods.
- "Good data scientists don't memorize — they derive."

Core Pillars



Together, these form the foundation for all data science workflows.

1. Linear Algebra

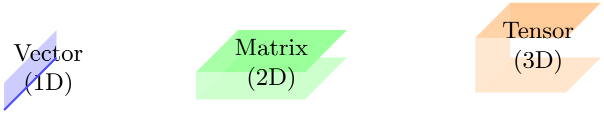
Key Concepts:

- **Vectors:** Ordered lists of numbers representing points or directions in space.
- **Matrices:** Two-dimensional arrays representing data or linear transformations.
- **Tensors:** Higher-dimensional generalizations of matrices, used in deep learning.
- **Core Operations:** Dot products, norms, projections, and orthogonality.
- **Eigenvalues and Eigenvectors:** Describe principal directions of data variance or transformation.
- **Singular Value Decomposition (SVD):** Decomposes a matrix into orthogonal components for compression and analysis.

Applications in Data Science:

- **Dimensionality Reduction:** Principal Component Analysis (PCA) uses eigenvectors to find major variance directions.
- **Word Embeddings:** Matrix factorization and SVD uncover latent relationships in language models.
- **Deep Learning:** Neural network weights and activations are represented as matrices and tensors.

Visualizing Vectors, Matrices, and Tensors



Interpretation: A vector is 1D, a matrix is 2D, and a tensor generalizes to 3D or more dimensions.

Core Operations, Eigenvectors, and SVD

Core Operations:

- **Dot Product:** A way to multiply two vectors. It measures how similar or aligned two directions are. Example: if two vectors point in the same direction, the dot product is large; if they are perpendicular, it is 0.
- **Norm:** The length (or size) of a vector — similar to distance in geometry. Shows how "long" or "strong" a vector is.
- **Orthogonality:** Means perpendicular or independent. Two vectors are orthogonal if their dot product is 0 (no overlap or correlation).

Eigenvalues and Eigenvectors:

- When a matrix transforms data (rotating, stretching, etc.), some directions stay the same — only their length changes.
- Those special directions are **eigenvectors**, and the amount of stretching or shrinking is the **eigenvalue**.
- In data science, they describe **principal directions of variance** — the main patterns or axes along which data spreads out (used in PCA).

Core Operations, Eigenvectors, and SVD

Singular Value Decomposition (SVD):

- A mathematical tool that breaks a matrix into simpler parts:

$$A = U\Sigma V^T$$

- U and V are orthogonal (like rotation axes), and Σ contains singular values (the strength of each direction).
- **Used for:**
  - **Compression:** Keep only the largest singular values to reduce data size (e.g., image compression).
  - **Noise Reduction:** Remove small singular values that represent noise.
  - **Recommendation Systems:** Find hidden relationships between users and items.

Understanding Linear Algebra Through Metaphors

Why use metaphors? Linear algebra describes shapes, movements, and patterns — metaphors help connect these to real-world intuition.

Key Metaphors:

- **Vector → Arrow or Direction:** A vector is like an arrow pointing somewhere — it has both *direction* and *length*. Example: wind blowing north-east at 20 km/h is a vector.
- **Matrix → Machine or Transformation:** A matrix acts like a machine that takes one arrow and turns it into another. It can rotate, stretch, flip, or compress vectors in space.
- **Dot Product → Conversation Between Vectors:** Measures how much two directions “agree.” If they point the same way, the result is large; if perpendicular, it’s zero.
- **Norm → Energy or Strength:** Think of the norm as the *energy* or *magnitude* of a vector — how strong or long the arrow is.
- **Eigenvectors → Stable Directions:** When the matrix-machine acts, most arrows change direction — but eigenvectors are the ones that stay pointing the same way, only stretched or shrunk.
- **Eigenvalues → Volume Knobs:** They tell how much each eigenvector is stretched — like turning up or down the volume on that direction.

2. Calculus & Optimization

Key Concepts:

- **Derivative:** Measures how a function changes — the *rate of change* or the *slope* of a curve.
- **Gradient:** A vector of partial derivatives that points in the direction of the steepest increase of a function.
- **Hessian:** A square matrix of second-order derivatives; captures how curvature (concavity/convexity) behaves.
- **Chain Rule:** Tells how to compute derivatives of composed functions — essential for backpropagation.
- **Partial Derivatives:** Measure how a function changes with respect to one variable, keeping others fixed.
- **Gradient Descent:** Iterative algorithm that moves in the opposite direction of the gradient to find minima.
- **Stochastic Gradient Descent (SGD):** Uses random subsets (mini-batches) of data to speed up optimization.
- **Convexity:** A property of functions where every local minimum is also a global minimum — makes optimization easier.
- **Saddle Points:** Points where the gradient is zero, but the function is neither a minimum nor a maximum.

Core Concepts in Calculus & Optimization

Concept	Definition	Metaphor / Intuition
Derivative	Measures how a function changes — the rate of change or slope.	The slope of a hill at one point.
Gradient	Vector of partial derivatives; points in the direction of steepest increase.	Arrow showing the steepest uphill path.
Hessian	Matrix of second-order derivatives; describes curvature and shape of a surface.	Detects how "bumpy" or "flat" the terrain is.
Chain Rule	Rule to compute derivatives of composed functions.	How change flows through linked processes.
Partial Derivative	Change in one variable while others stay fixed.	Adjusting one knob at a time on a control panel.
Gradient Descent	Iterative method moving opposite to gradient to reach a minimum.	Rolling a ball downhill to the lowest point.
Stochastic Gradient Descent (SGD)	Uses random data batches to update parameters efficiently.	Taking noisy but quicker downhill steps.
Convexity	Property where every local minimum is a global minimum.	A single bowl — no hidden dips or traps.
Saddle Point	Gradient is zero, but not a min or max.	A mountain pass — goes up one way, down another.

Summary: Calculus tells us *how things change*; optimization tells us *how to make them better.*

2. Calculus & Optimization

Applications in Data Science:

- **Training ML Models:** Optimization algorithms minimize the loss function to improve model accuracy.
- **Backpropagation:** Uses the chain rule to efficiently compute gradients through layers in deep networks.
- **Regularization:** Adds penalty terms to control model complexity and enforce smoothness.
- **Hyperparameter Tuning:** Optimization helps find the best learning rate, weights, or architecture parameters.

In essence: Calculus provides the language of change, and optimization turns it into a method for learning from data.

3. Probability & Statistics

Core Topics:

- **Random Variables:** Quantities whose values result from random phenomena (e.g., dice rolls, measurements).
- **Distributions:** Describe how probabilities are spread across possible outcomes (Normal, Binomial, Poisson, etc.).
- **Expectation:** The long-run average or "center" of a random variable, $E[X]$.
- **Variance & Covariance:** Measure variability ($Var[X]$) and joint variability between two variables ($Cov[X, Y]$).
- **Sampling:** Selecting a subset of data to estimate characteristics of the full population.
- **Hypothesis Testing:** Evaluates whether observed data support or reject a proposed claim.

3. Probability & Statistics

Applications in Data Science:

- **Model Uncertainty & Inference:** Quantify confidence in predictions and parameters.
- **Feature Selection:** Use correlation or hypothesis tests to determine significant variables.
- **Probabilistic Models:** Naive Bayes, Hidden Markov Models (HMMs), and Bayesian networks.
- **Confidence Intervals:** Assess performance differences between models or treatments.

In essence: Probability models uncertainty; statistics helps us learn from data and make reliable inferences.

Explanation: Confidence Intervals

Confidence Intervals (CI):

- A **confidence interval** gives a range of plausible values for an unknown quantity (e.g., a model's accuracy or a population mean).
- Instead of stating a single estimate such as:

$$\text{Accuracy} = 0.85,$$

  we report a range that reflects uncertainty:

$$95\% \text{ Confidence Interval: } [0.82, 0.88].$$

- Interpretation: If we repeated the experiment many times, about 95% of those intervals would contain the true value.
- It expresses **sampling uncertainty** — wider intervals mean more uncertainty in the estimate.

In short: Confidence intervals tell us how certain (or uncertain) we are about our estimates.

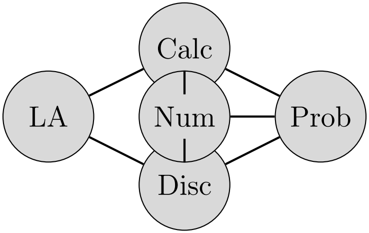
4. Discrete Math & Logic

Essential Areas:

- **Sets:** Collections of unique elements; foundation for reasoning about data, membership, and relations.
- **Functions:** Mappings between inputs and outputs — used to define transformations in algorithms.
- **Combinatorics:** Counting methods to measure possible outcomes or configurations (e.g., feature combinations).
- **Graph Theory:** Study of nodes and edges — models relationships and networks (e.g., social graphs, knowledge graphs).
- **Trees:** Hierarchical graphs used in decision trees, search algorithms, and file systems.
- **Boolean Logic:** Reasoning using True/False values; forms the basis of conditionals and decision rules.
- **Logic Gates:** Hardware or software implementations of Boolean functions (AND, OR, NOT, XOR).
- **Complexity:** Study of how computation time or memory grows with input size ($O(n)$, $O(n^2)$, etc.).
- **Algorithm Analysis:** Evaluates efficiency, correctness, and scalability of computational methods.

Putting It All Together

- **Linear Algebra** — data representation.
- **Calculus** — optimization and learning.
- **Probability & Stats** — uncertainty and inference.
- **Discrete Math** — logic, structures, algorithms..



Practical Study Roadmap

- Start with **Linear Algebra** and **Probability** — they show up everywhere.
- Add **Calculus** — to understand optimization and deep learning.
- Learn **Statistics** — to make sound inferences and decisions.
- Explore **Discrete Math** — for graph theory and algorithmic thinking.
- Practice **Numerical Computation** — implement algorithms from scratch using Python and NumPy.

Goal: Not to memorize formulas, but to build true mathematical *intuition.*

Summary: Math Mindset for Data Scientists

- Understand the assumptions behind your models.
- Question distributions, stability, and convergence.
- Simplify complex ideas into geometric intuition.
- Combine theory with computation and curiosity.

"Math is not a prerequisite to do data science — it's how you do it well."