An Introduction to Probability

First Probability Ideas and First Steps in R

Martin Summer

11 January, 2026

First Probability Ideas and First Steps in R

Introduction

- Finance inherently involves **uncertainty** and **risk**, making probability essential.
- Finance focuses on allocating and pricing money over time:
  - Saving defers consumption for future purchases.
  - Borrowing enables investment today with future repayment.
- The future is uncertain and unpredictable, but **probability theory** provides tools to:
  - Quantify and analyze uncertainty.
  - Manage financial risks effectively.

Why Study Probability?

- Probability is fundamental to managing uncertainty in Finance.
- Originated as a mathematical theory in the **16th and 17th century**:
  - Influenced by gambling debates.
  - Scholars like **Cardano**, **Pascal**, **Fermat**, and **Bernoulli** contributed.
- Humans have long been aware of chance (e.g., gambling, goddess of chance), but only later sought to **measure uncertainty mathematically**.

First Steps in Probability

A Simple Starting Point

- We'll begin with a **classical probability model**: Tossing a coin.
- This example helps us:
  - Understand fundamental concepts in probability.
  - Explore basic R programming.

Tossing a Fair Coin: First Probability Ideas

- A **fair coin** has an equal chance of showing Heads or Tails.
- When tossed, the outcome is uncertain: Heads or Tails.
- Visualizing the process:

Figure 1: Figure 1: Tossing a coin

How Coin Tossing Matters for Finance

Simulating a coin toss introduces key concepts:

- Equal probabilities for outcomes.
- Link to Finance: Imagine a stock investment with a 50% chance of gain or loss.
- Builds intuition for probability and computational Finance.

The Role and Nature of Idealizations

- **Idealizations** are standard in probability and essential in Finance.
- Example: Stock price movement as a conceptual random experiment.
  - Price cannot fall below 0.
  - Hard to define the highest possible price.

Sample Space of Stock Prices

- Common assumption: Sample space is the entire interval of non-negative real numbers:
  - $S = [0, \infty)$.

Why Use Idealizations?

- Many models in Finance assume **arbitrarily high prices**:
  - Prices can rise without bound, but with **arbitrarily small probabilities** as they increase.
- Practical reasoning:
  - Imposing an upper bound x may seem reasonable, but it introduces awkward assumptions:
    - Why can't the price be just a cent higher than x?
  - Simplicity and convenience outweigh strict realism.

Key Takeaway

- Idealizations simplify models without significant practical harm, enabling tractable analysis in Finance.

Classical Probability: Measuring Uncertainty

- **Probability** quantifies how likely an event is to occur.
- Measurement idea:
  - Similar to measuring length: Choose a standard and count.
  - Measure probability by counting equally probable cases.

Formula for Classical Probability

$$P(A) = \frac{\text{Number of cases where } A \text{ occurs}}{\text{Total number of cases}}$$

- Example: Tossing a fair coin
  - Heads: $P(\text{Heads}) = \frac{1}{2}$ (1 favorable case out of 2 total cases).

Properties of Classical Probability

1. Probability is never negative: $P(A) \geq 0$
2. If an event A occurs in all cases: $P(A) = 1$
3. For mutually exclusive events A, B: $P(A \text{ or } B) = P(A) + P(B)$

Example for Rule 3

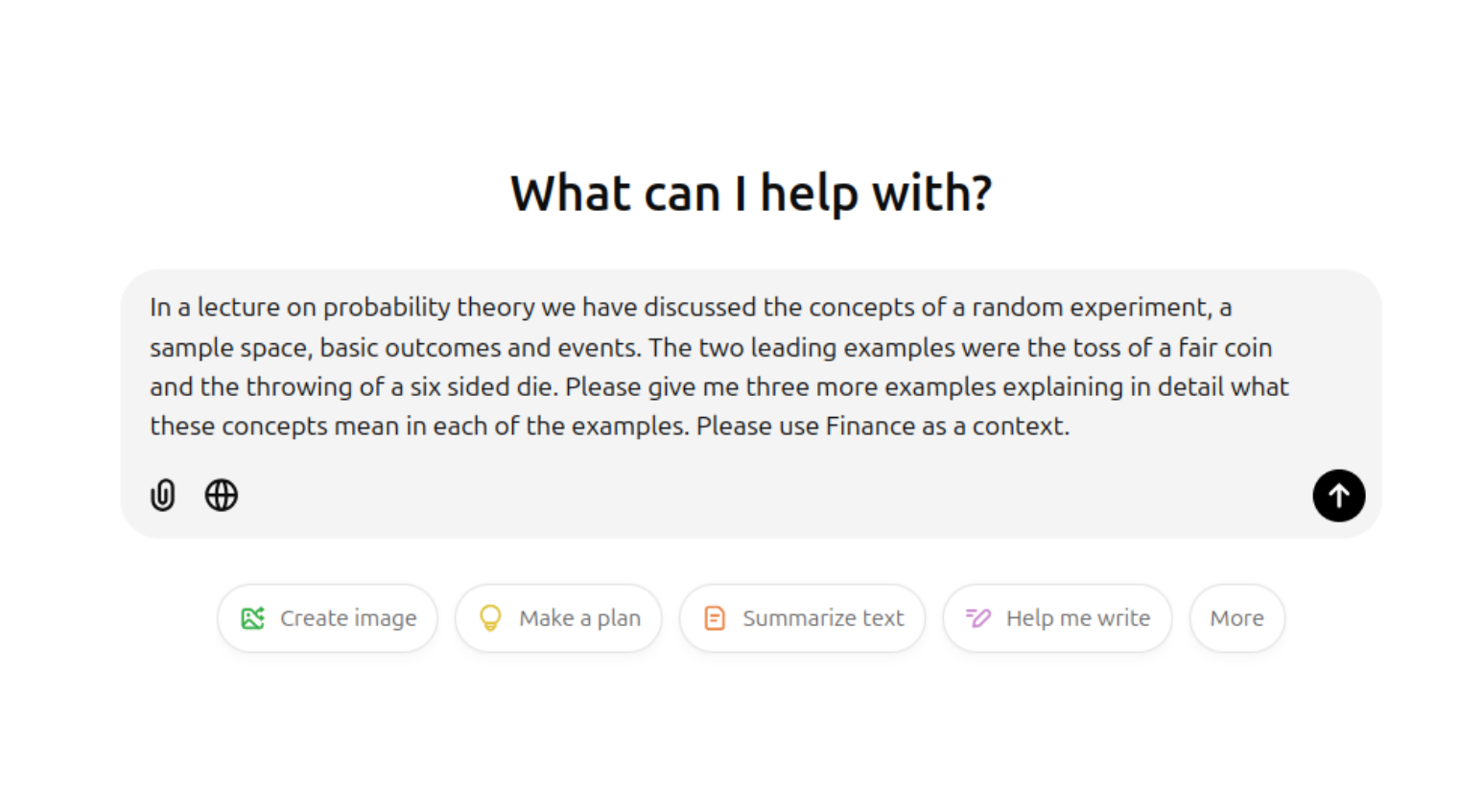
- Drawing a King (A) or a Queen (B) from a deck of 52 cards:
  - $P(A) = \frac{4}{52}, P(B) = \frac{4}{52}$.
  - Mutually exclusive events: A and B cannot occur together.
  - $P(A \text{ or } B) = \frac{4}{52} + \frac{4}{52} = \frac{8}{52} = \frac{2}{13}$.

Complementary Events

- Probability of an event **not** occurring: $P(\text{not } A) = 1 - P(A)$

Leveraging LLMs for Learning

- Enhance understanding with **large language models (LLMs)** like ChatGPT, Claude or Gemini.
- Example prompt for ChatGPT:
  - Explore random experiments, sample spaces, basic outcomes, and events in Finance.



Transition to R: Making It Tangible

- To apply these ideas practically, we turn to R.
- Coin tossing example revisited:
    - Learn basic R concepts.
    - Simulate probability experiments programmatically.

Tossing a Coin on the Computer: First Steps in R

The R User Interface

- RStudio allows us to interact with the computer and run R commands.
- Launch RStudio to see a screen similar to this:

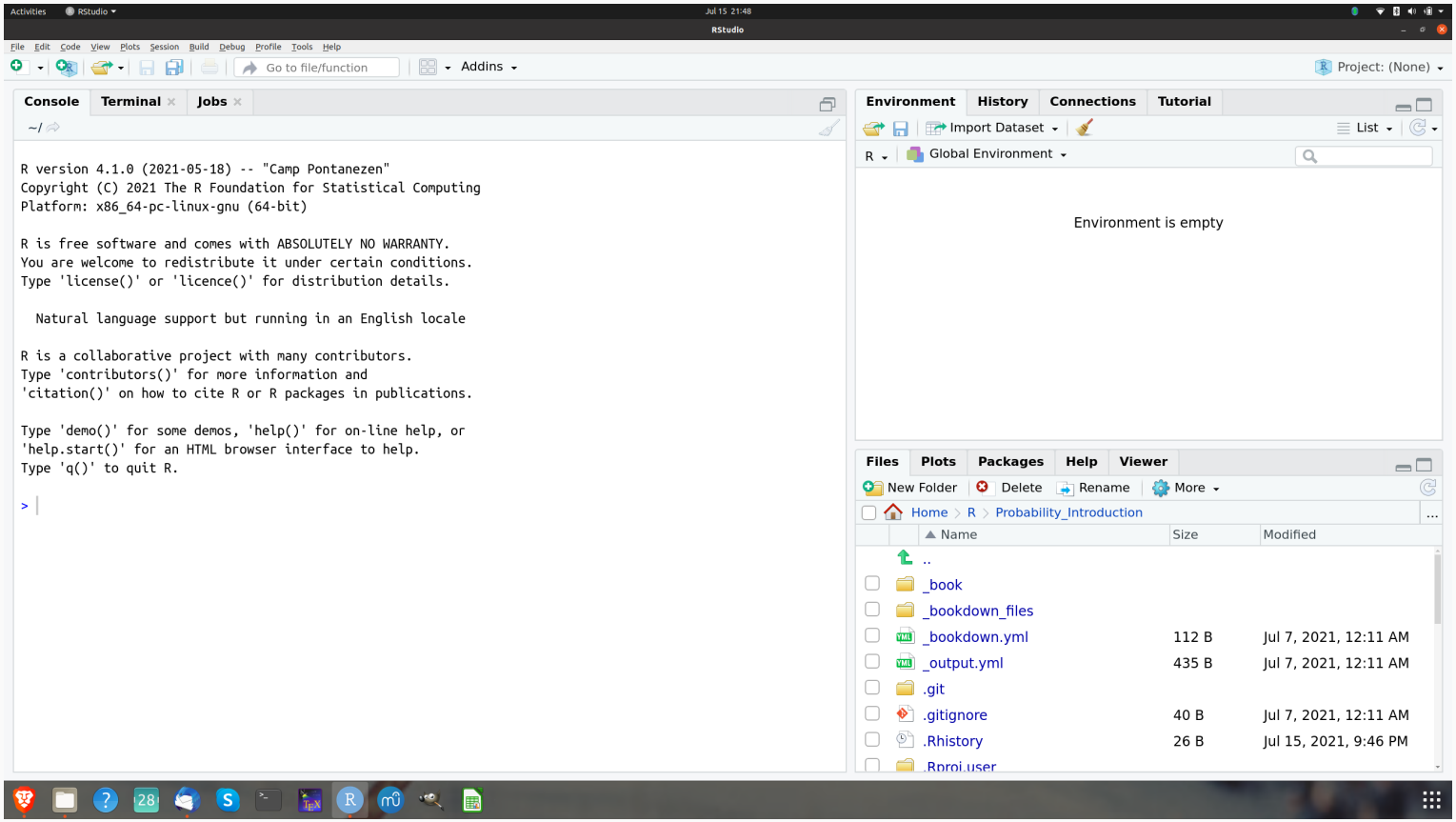


Figure 3: The RStudio startup screen

Interacting with RStudio

- RStudio provides an easy way to interact with R.
- Commands are typed at the **prompt** (>) in the **Console** pane.
- Press **Enter** to send the command to R.
- R processes the command and displays the result with a new prompt.

```
1 + 1
```

```
[1] 2
```

- Code chunks are light-gray boxes containing R commands, e.g., `1 + 1`.
- The R prompt (>) is not displayed in code chunks; it appears only in the Console.
- `[1]` means that this is the first element in the output.

Understanding the Colon Operator (`:`)

- Entering the command below lists all integers from 20 to 60:

```
20:60
```

```
 [1] 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38
[26] 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60
```

- The result begins with `[1]`, indicating the first value (`20`).
- Line breaks occur when values exceed the line length. For example, `[26]` marks the 26th value (`45`).
- The colon operator (`:`) creates integer sequences between two numbers and is frequently used in R.

Incomplete Commands in R

- R requires a complete command to execute when the return key is pressed.
- If the command is incomplete, R displays a `+` instead of a new prompt.
- Example:

```
5*
+ 4
```

- Only when you close the expression by entering `4` R will output `20`.

Understanding Error Messages in R

- R returns an error message if it does not understand a command.
- Error messages help you identify and debug issues.
- Example of an invalid command:

```
5%3
```

- Here you will get an error message like: `Error: unexpected input in "5%3"`

Dealing with Errors in R

- Some errors are easy to understand, such as R not knowing how to handle **%** in this context.
- Other errors might be less obvious and require further investigation.

Strategies for Resolving Errors

- Search the error message online. Many others may have encountered the same problem.
- A highly recommended resource:
  - Stack Overflow for R-related questions.
- Ask a Large Language Model (LLM) like ChatGPT for help.
  - LLMs often provide useful explanations and solutions.

Creating a Coin in R

- A coin has two possible outcomes: Heads or Tails.
- Encode Heads as `1` and Tails as `0`.
- Use the colon operator `:` to create a range of numbers from 0 to 1.

```
0:1
```

```
[1] 0 1
```

Probability Concepts Using a Coin Toss

- **Basic outcome**: A specific result of the coin toss, e.g., `1` for Heads.
- **Sample space**: The set of all possible outcomes:
  - $S = \{0, 1\}$
  - Often denoted as Ω in probability texts.
- **Random experiment**: Tossing the coin, where the outcome is uncertain but can be precisely determined after the toss.
- **Empty set**: The event that an impossible outcome occurs, e.g., the outcome that the coin toss result is 2. In this case $A = \emptyset$.
  - $\emptyset$: A set containing no elements, as a coin toss cannot produce anything other than `0` or `1`.

Understanding Objects in R

- In R, you can save data by storing it in **objects**.
- An object is a name you assign to store data.
- Example: Store the value `1` in an object named `Heads`.

```
Heads <- 1
```

Referring to Object Values

- Once stored in an object, you can use the object's name to perform operations.
- Example: Divide `Heads` by `2` to get a meaningful result.

```
Heads / 2
```

```
[1] 0.5
```

Storing a Coin in an Object

- To make the `coin` more useful, store it in an R object.
- Create an object named `coin` and assign the vector `0,1` to it:

```
coin <- 0:1
```

Tracking Objects in the RStudio Environment Pane

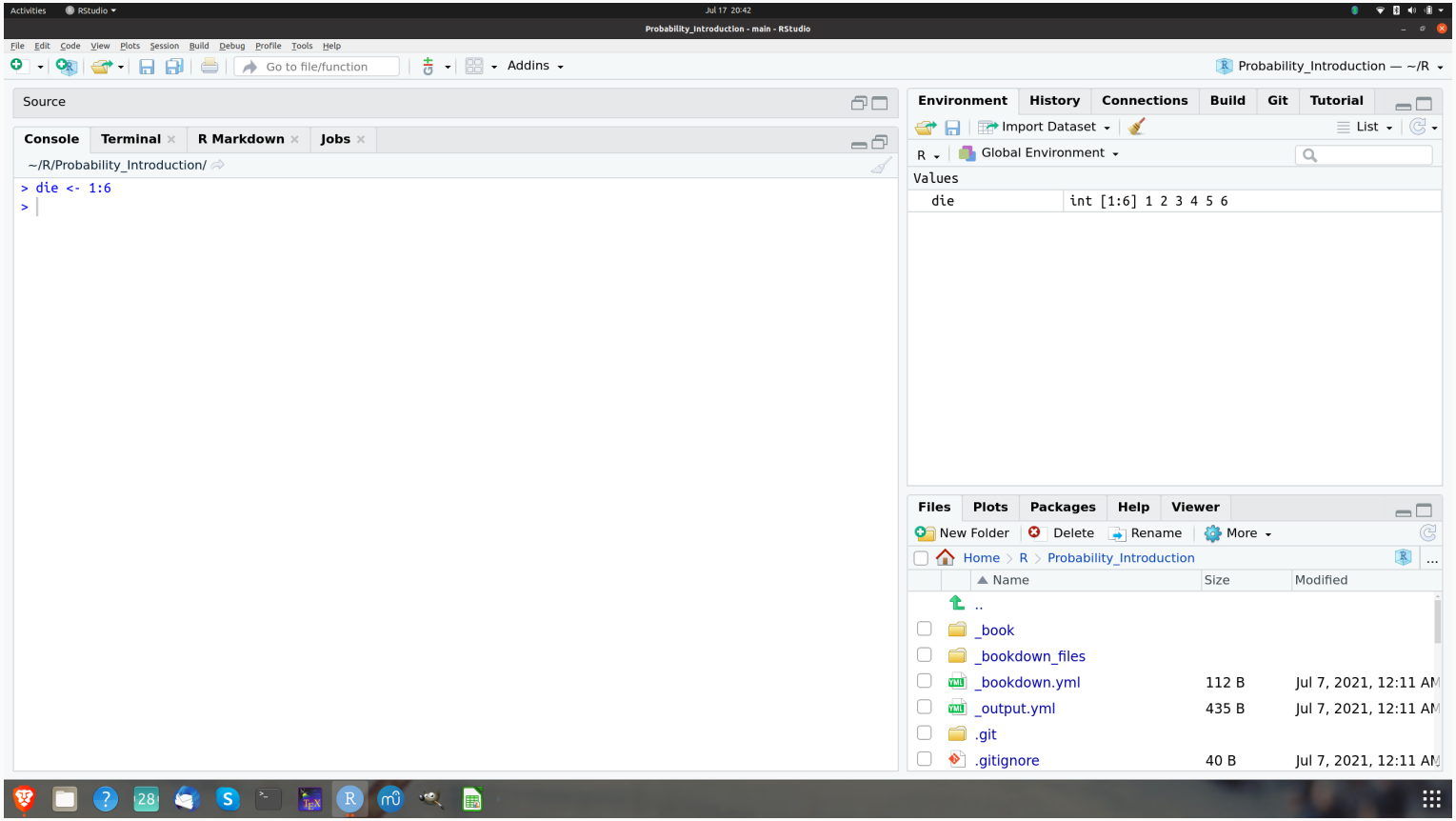


Figure 4: The RStudio Environment pane keeps track of the objects you have created

Naming and Managing Objects in R

- Object naming rules:
  - Must not start with a number.
  - Special symbols like `^, !, $, @, +, -, /, *` cannot be used.
  - R is case-sensitive: `object` and `Object` are different.
- Overwriting objects:
  - Assigning a new value to an existing object overwrites it without warning.
- Viewing created objects:
  - Use the **Environment pane** in RStudio.
  - Alternatively, type `ls()` at the prompt to list all objects.

Element-Wise Operations in R

- Using `*` performs element-wise multiplication for vectors.
- Example:

```
coin * coin
```

```
[1] 0 1
```

- This differs from linear algebra, where vector multiplication requires an inner product.
- To compute an inner product, use %*%:

```
coin %*% coin
```

```
     [,1]
[1,]    1
```

.

Element-Wise Execution

- Element-wise execution applies operations to each component of a vector.
- Example: Adding `1` to each element in `coin`.

```
coin + 1
```

```
[1] 1 2
```

Understanding Recycling in R

- R repeats shorter vectors to match the length of longer vectors in operations.
- Example:

```
coin + 1
```

```
[1] 1 2
```

Recycling Rule and Warnings

- Example of recycling with non-matching lengths:

```
coin + 1:4
```

```
[1] 1 3 3 5
```

- As of R 4.2.0 (April 2022):
  - Warnings for non-multiple lengths are removed.
  - This reduces noise but requires users to be cautious about unintended recycling.
-

Understanding Functions in R

- Functions are used to manipulate data and perform computations.
- Syntax: `function_name(argument)`
- Example:

```
sqrt(4)
```

```
[1] 2
```

- Arguments are the data passed to a function.
- Arguments can include raw data, R objects, or results from other functions.

Example of Function Evaluation

```
numbers <- 1:7
mean(numbers)
```

```
[1] 4
```

```
round(mean(numbers))
```

```
[1] 4
```

- `mean(numbers)` calculates the mean.
- `round(mean(numbers))` rounds the result to the nearest integer.

Simulating Random Experiments with `sample()`

- The `sample()` function randomly selects elements from a vector.
- Arguments:
  - `x`: The vector to sample from.
  - `size`: The number of elements to return.
- Example:

```
sample(x = 0:1, size = 1)
```

```
[1] 0
```

Using Function Arguments

- Every function argument has a name.
  - Assign values with `name = value`.
- Example: Named arguments for clarity.

```
sample(x = coin, size = 1)
```

```
[1] 0
```

- Argument names are optional. R matches arguments by order if names are omitted.

Checking Arguments

- Use `args()` to check argument names and defaults

```
args(round)
```

```
function (x, digits = 0, ...)
NULL
```

- Example: The `round()` function has an optional digits argument with a default value of 0.
- Writing argument names is recommended for clear, error-free code.

Writing Your Own Functions in R

- A function in R has three components:
  - **Name**: The function's identifier.
  - **Arguments**: Input values for the function.
  - **Function body**: The code the function executes.
- Syntax for creating a function:

```
my_function <- function() {}
```

A Function to Toss a Coin

```r
toss_coin <- function() {
  coin <- 0:1
  sample(coin, size = 1)
}
```

Using Your Function

- Call the function with parentheses: `toss_coin()`.
- Example: Toss the coin multiple times:

```
toss_coin()
```

```
[1] 0
```

```
toss_coin()
```

```
[1] 0
```

```
toss_coin()
```

```
[1] 0
```

Exploring the `sample()` Function

- View the arguments of the `sample` function:

```
args(sample)
```

```
function (x, size, replace = FALSE, prob = NULL)
NULL
```

- Focus on the `replace` argument:
  - Default: replace = FALSE (no replacement).
  - This means each element can only be drawn once.
  - Example: Tossing two coins with replace = FALSE makes it impossible for both to show Heads.

Enabling Replacement in Sampling

- To allow duplicate outcomes (e.g., two Heads), set `replace = TRUE`.

```
sample(x = coin, size = 2, replace = TRUE)
```

```
[1] 0 0
```

- This behavior enables proper modeling of random experiments, such as tossing multiple coins.
- Remember: The parentheses `( )` trigger function execution.
- Code in the function body runs sequentially, and the result of the last line is returned.

Function Arguments and Errors

- If an object used inside a function is not defined, R will throw an error.
- Example: Missing the `ball` object in `toss_coin2`:

```
toss_coin2 <- function() {
  sample(ball, size = 1)
}
```

- Calling the function results in: `Error in sample(ball, size = 1) : object 'ball' not found`

Adding an Argument to the Function

- Define `ball` as an argument:

```
toss_coin2 <- function(ball) {
  sample(ball, size = 1)
}
```

- Now the function works if we supply ball when calling it

```
toss_coin2(ball = 0:1)
```

```
[1] 0
```

Default Arguments in Functions

- Provide a default value for `ball`:

```
toss_coin2 <- function(ball = 0:1) {
  sample(ball, size = 1)
}
```

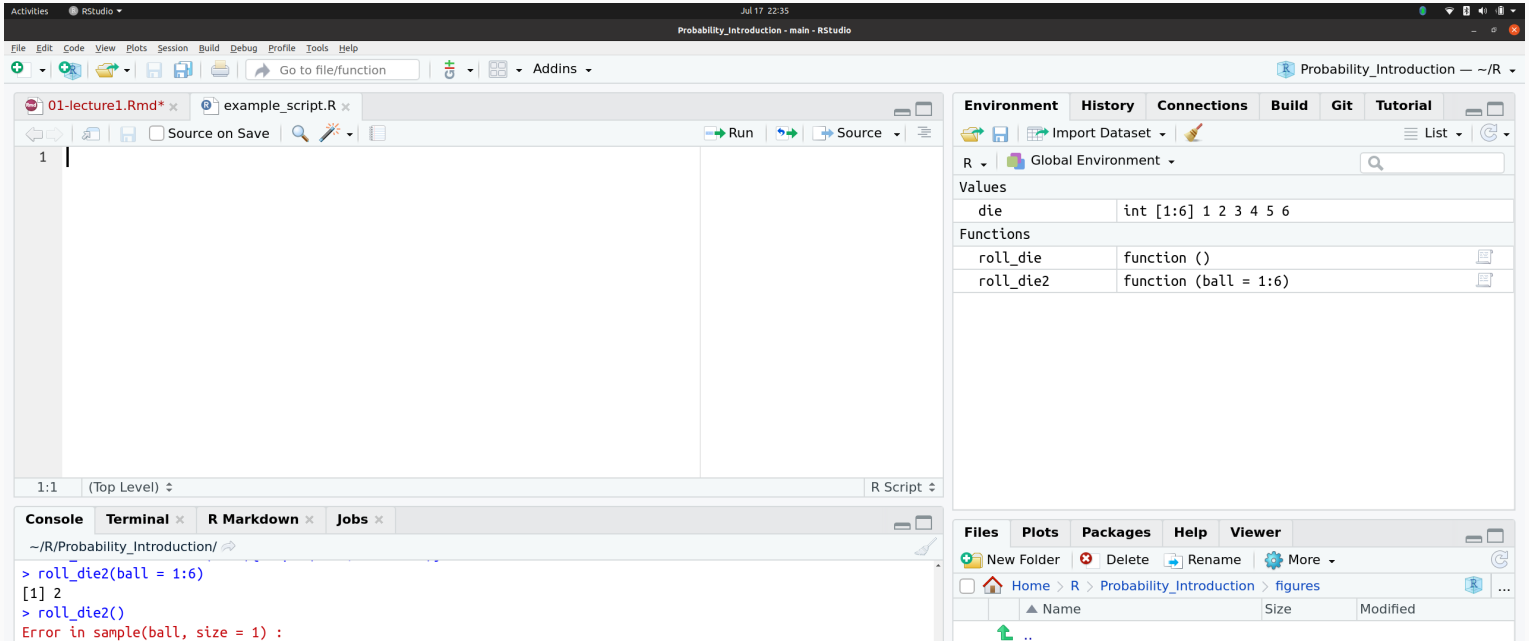
- Now calling the function without argument works as intended:

```
toss_coin2()
```

```
[1] 1
```

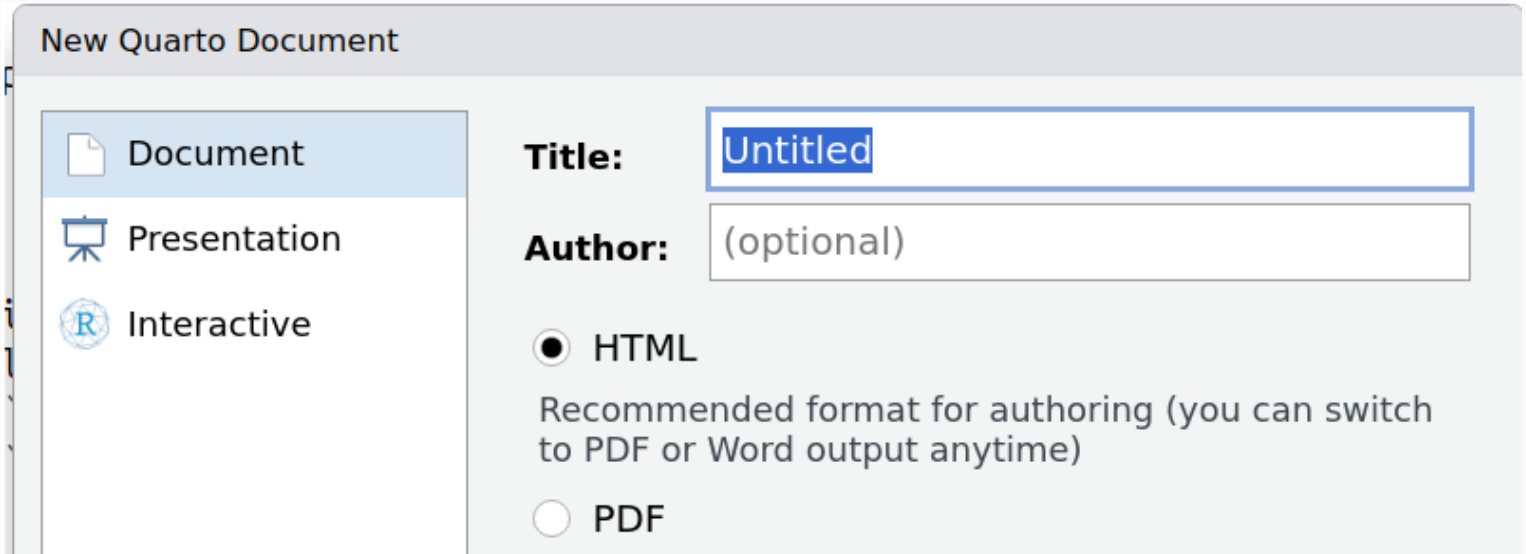
Using Scripts in RStudio

- A **script** is a draft where you can write, save, and edit your R code.
- Advantages of using scripts:
  - Keeps your code organized.
  - Enables reproducibility.
  - Easy to edit and share.
- Create a script:
  - `File > New File > R Script` in the RStudio menu.
  - Save the script via `File > Save As`.



Introduction to Quarto Documents

- Quarto combines explanatory text, R code, and output into one cohesive document.
- Uses:
  - Documenting your learning process.
  - Conducting reproducible research.
  - Preparing assignments.
- Create a Quarto document:
  - `File > New File > Quarto Document...`
  - Choose `HTML` format for easy preview.



Adding and Running Code in Quarto

- Add text and code chunks to explain and experiment.

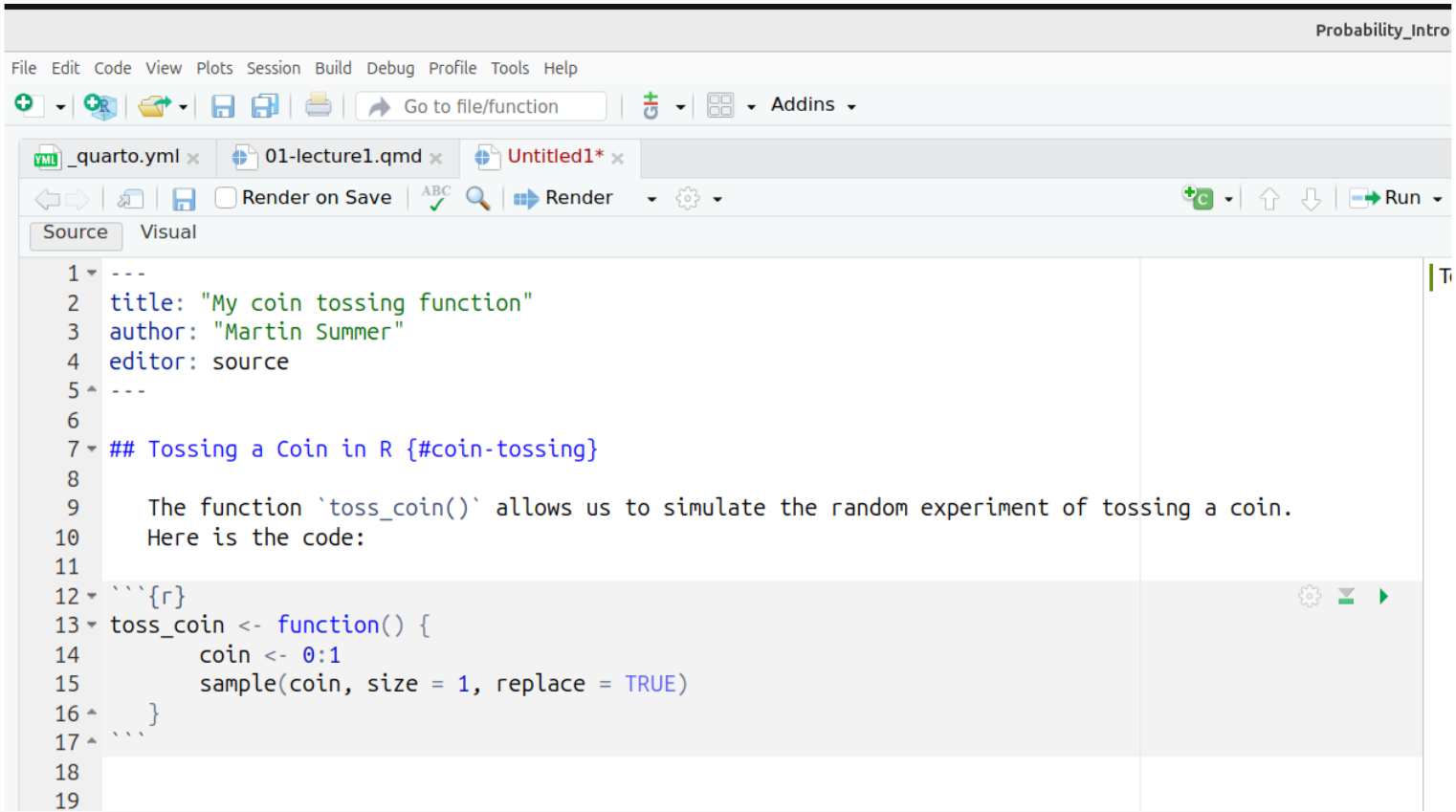


Figure 7: Combining text and code in Quarto documents

Enhancing Quarto Documents

- Lecture notes and slides are written in Quarto.
- Quarto supports other programming languages like Python and Julia.
- Explore more at https://quarto.org/.
- Advantages:
    - Create interactive, well-documented code files.
    -

Using Packages and Finding Help

- Our function `toss_coin()` simulates a coin toss.
- To test if the coin is *fair*, we use:
  - **Repetition**: Tossing the coin multiple times.
  - **Visualization**: Tools to display the results.
- R's capabilities are extended through **packages**:
  - Packages are add-ons that provide new functions.
  -

Installing and Loading Packages

- Example: The **ggplot2** package for visualization.
- Install the package (requires internet):

```
install.packages("ggplot2")
```

- load package to use its functions

```
library(ggplot2)
```

Simulating Multiple Coin Tosses

- Use the `replicate()` function to repeat a command multiple times.
- Example: Toss a coin 100 times and save the results:

```
set.seed(123)
tosses <- replicate(100, toss_coin())
```

Visualizing Coin Toss Results

```
# Create a data frame for plotting
tosses_df <- data.frame(
  Outcome = factor(tosses, levels = c(0, 1),
                   labels = c("Tails", "Heads"))
)

# Plot the results using ggplot2
ggplot(tosses_df, aes(x = Outcome)) +
  geom_bar(fill = "skyblue", color = "black") +
  labs(
    title = "Occurrences of Heads and Tails",
    x = "Outcome",
    y = "Count"
  ) +
  theme_minimal()
```

Visualizing Coin Toss Results: 100 tosses

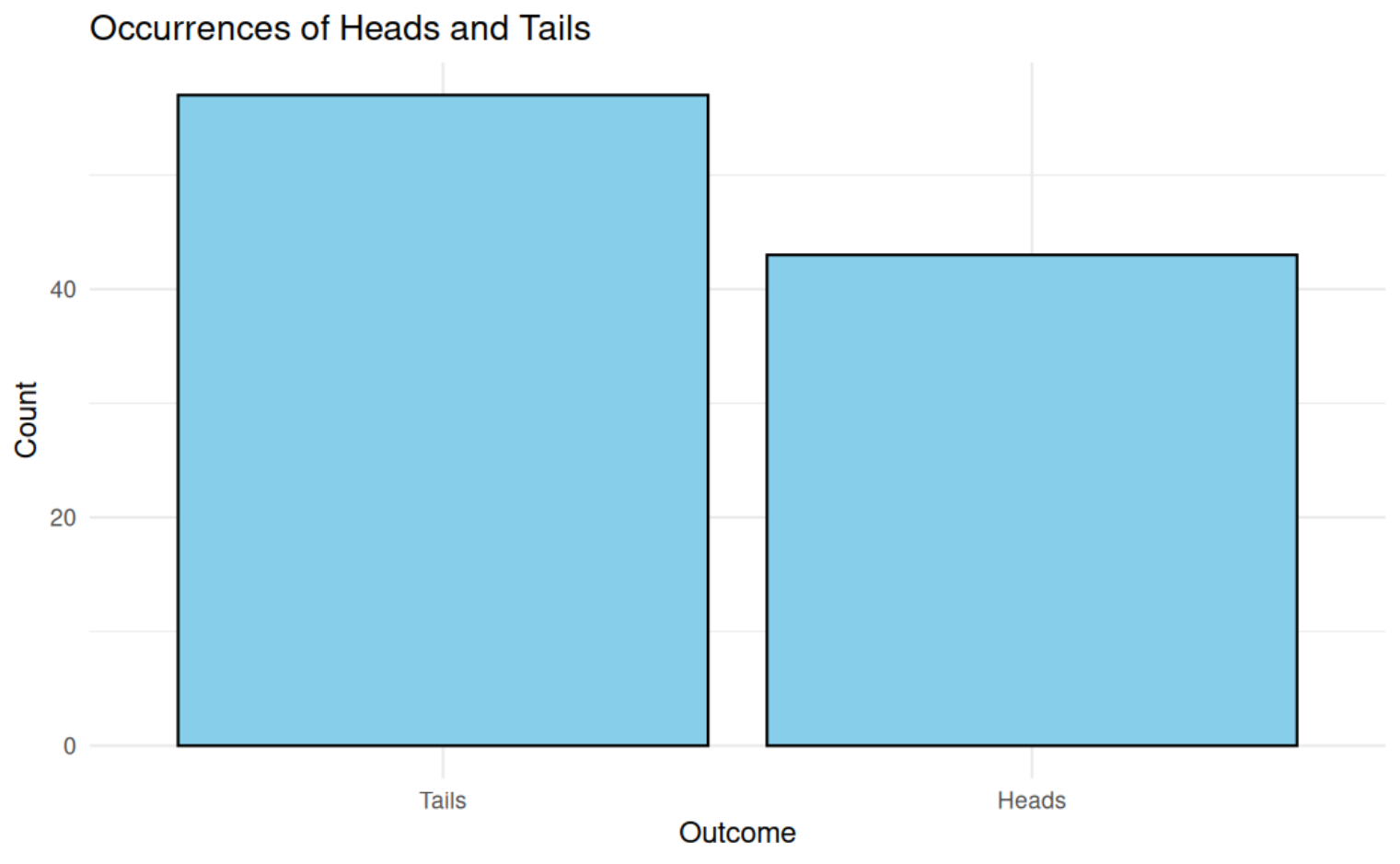


Figure 8: Coin tossed 100 times

Visualizing Coin Toss Results

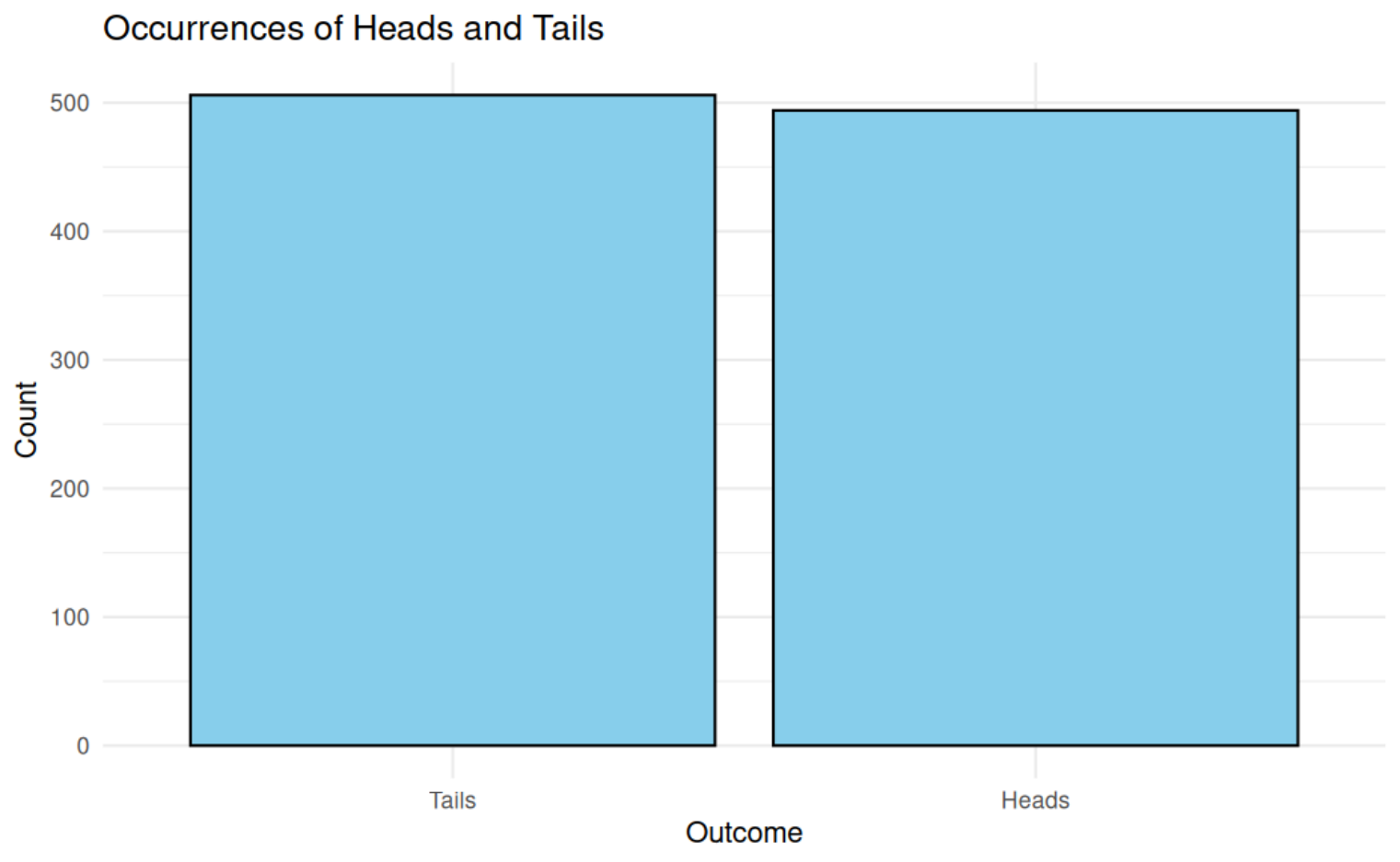


Figure 9: Coin tossed 1000 times

The Relative Frequency Notion of Probability

- The visualization shows that each face comes up approximately 500 times.
- This suggests there is no significant evidence that our virtual die is loaded.
- We have transitioned from classical probability (counting equally probable cases) to:
    - **Relative frequency notion of probability**:
        - Repeating random experiments to assess probabilities by visualizing outcome frequencies.
- Key takeaway:
    - Practitioners have historically assessed probabilities by observing event frequencies.
    - In the next lecture, we will formally explore the connection between these two concepts and why this approach works.

Tip for R Novices: Leveraging LLMs for Learning

- Use Large Language Models (LLMs) to enhance your learning experience.
- Examples of how to use LLMs:
  - **Translating Code**: If you know Python, ask the LLM to translate R code into Python and compare.
  - **Understanding Code**:
    - Ask the LLM to explain R code, such as the `ggplot2` plotting function.
    - You will likely receive clear and detailed explanations.
- Tip: Experiment with the LLM for explanations or code translations to deepen your understanding.
- **Try it and see how it helps!**

Getting Help in R

- Every R function comes with detailed documentation and a help page.
- To access the help page for a function, type a question mark **?** followed by the function name at the prompt:

```
?sample
```

Structure of an R Help Page

- The help page has a consistent structure across all R functions:
  1. **Name and Package**: The function name and its source package (e.g., `sample` in `{base}`).
  2. **Description**: A short summary of what the function does.
  3. **Usage**: Function syntax with arguments.

Exploring Arguments in Help Pages

- The **Arguments** section explains the function's inputs:
  - `x`: A vector or a positive integer to sample from.
  - `size`: Number of items to choose.
  - `replace`: Whether sampling is with replacement.
  - `prob`: Probability weights for sampling.
- Fields provide additional information:
  1. **Details**: In-depth description of the function.
  2. **Value**: What the function returns.
  3. **See Also**: Related R functions.
  4. **Examples**: Ready-to-run example code demonstrating practical use.

Applications

The Birthday Problem

- **Question**: What is the probability that at least two people in a room share the same birthday?
    - Assumptions:
        - Birthdays are equally likely across 365 days.
        - No leap years or twins.
        - Birthdays are independent.
- **Significance**: Beyond recreational math:
    - Highlights the probability of coincidences.
    - Applications in cryptography and blockchain security.
- Approach:
    - Frame the problem mathematically.
    - Compute probabilities using R.
    -

Sample Space in the Birthday Problem

- The sample space contains all possible combinations of birthdays for a group:
  - For n people, the sample space size is 365^n.
  - Example for $n = 3$:

$$(1, 1, 1), (1, 1, 2), ..., (365, 365, 365)$$

- Mathematically: $S = \{x | x \in 365^n\}$
  - Cartesian product: Ordered n-tuples from 365^n.

Complement Rule and Birthday Coincidence

- Use the complement rule:
  - $P(\text{at least two share a birthday}) = 1 - P(\text{no one shares a birthday})$
- Compute $P(\text{no one shares a birthday})$:
  1. First person: $(365/365)$.
  2. Second person: $(364/365)$.
  3. Third person: $(363/365)$.
  4. Continue for n people.
- Final probability:

$$P(\text{at least two share a birthday}) = 1 - \frac{365 \times 364 \times \cdots \times (365 - n + 1)}{365^n}$$

Surprising Result for $n = 23$

- For $n = 23$ people, calculate the probability using R:

```
1 - prod(365:343)/365^23
```

```
[1] 0.5072972
```

- Tools used:
  - `prod()`: Computes the product of a vector.
  - `^`: Raises numbers to a power.
  - `:` Creates descending lists (e.g., 365:343).

Result: With 23 people, the probability of at least one shared birthday exceeds 50%.

With 50 people, the probability is over 97%.

Visualizing the Birthday Problem

- Let's plot how the probability grows with the size of the group.
- Using **base R** for visualization (not **ggplot2**):

```r
# Create a group size vector
group_sizes <- 1:50
# Compute probabilities
probabilities <- sapply(group_sizes, function(n) {
  1 - prod(365:(365-n+1))/365^n
})
```

Visualizing the Birthday Problem

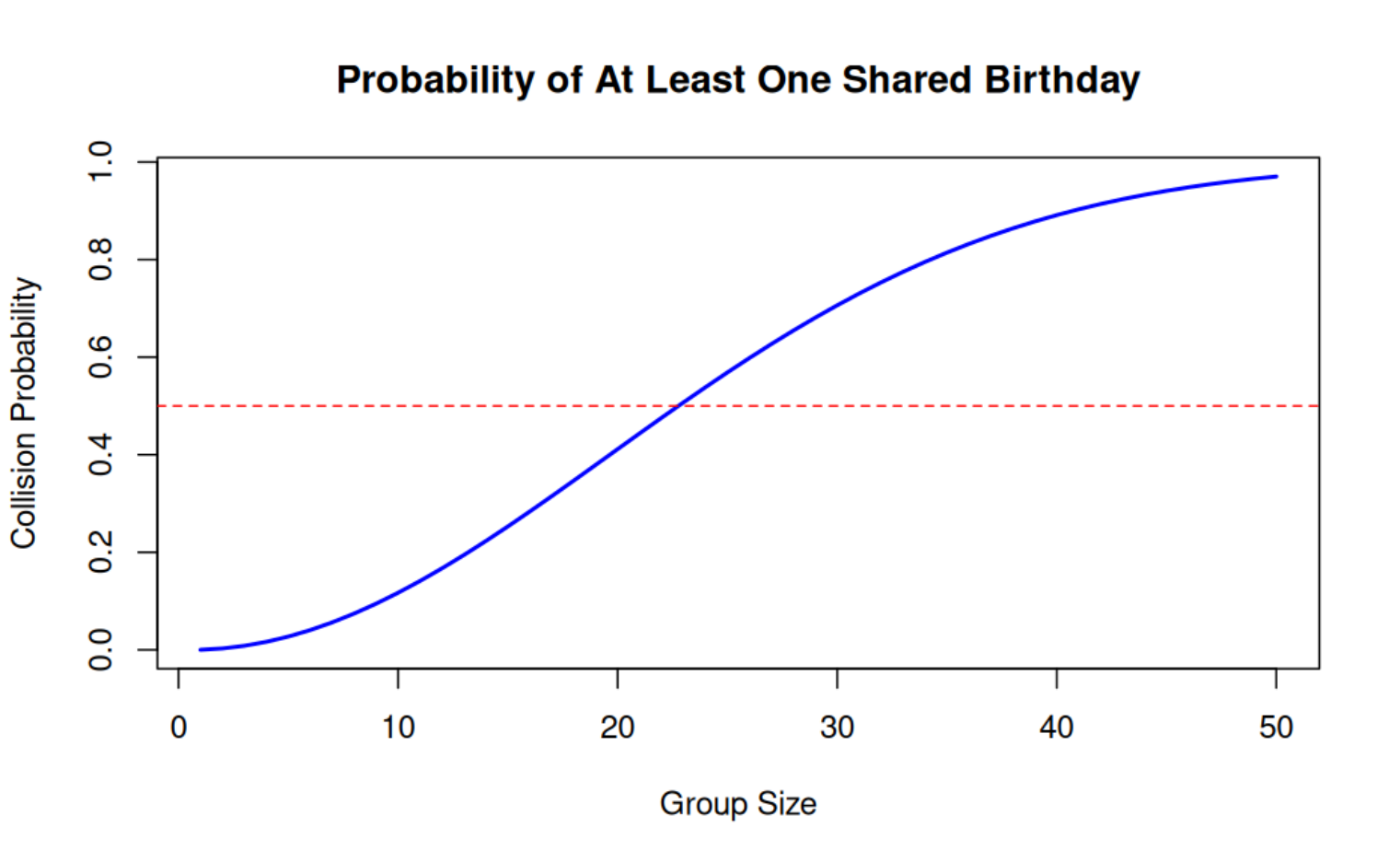


Figure 10: Probability of at least one shared birthday

Determining birthday collisions by simulation

Instead of deriving the probability analytically, we can simulate the birthday experiment if we use the relative frequency notion of probability which we informally had used in our visualization of coin tosses. Here's how:

1. Generate random birthdays for n people using `sample(1:365, n, replace = TRUE)`.
2. Count how often at least two people share a birthday using the `duplicated() function`in combination with the logical function `any()`.
3. Repeat the simulation many times (e.g., 10,000) using replicate() to estimate the probability.

Determining birthday collisions by simulation

```r
sim_birthday <- function(n, trials = 10000) {
  results <- replicate(trials, {
    birthdays <- sample(1:365, n, replace = TRUE)
    any(duplicated(birthdays))
  })
  mean(results)
}
```

Step-by-Step Breakdown of `sim_birthday` Function

1. **Defining the Function**:
   - `sim_birthday(n, trials = 10000)`
     - `n`: Number of people in the room (group size).
     - `trials`: Number of simulation repetitions (default: 10,000).
   - Parameters allow customization for different group sizes and precision levels.
2. **Using `replicate()`**:
   - Repeats the experiment `trials` times.
   -

Simulating Birthdays and Checking for Duplicates

3. Simulating Birthdays:
    - `sample()` generates n random birthdays (1 to 365) with replacement.
    - Assumptions:
        - Birthdays are independent and uniformly distributed.
        - Sampling with replacement allows for shared birthdays.
4. Checking for Duplicates:
    - `duplicated()` identifies repeated birthdays.
    - `any()` checks if there is at least one duplicate.
    -

Calculating the Probability

5. Calculating the Probability:
    - `results` vector stores TRUE (collision) or FALSE (no collision) for each trial.
    - `mean(results)` calculates the proportion of trials with a collision:
        - Probability = Favorable outcomes / Total trials.
    - This step connects to the relative frequency definition of probability:
        - Probability is the ratio of favorable outcomes to total trials.

Simulating the Birthday Problem

- Compute the birthday collision probability for:
  - $n = 23$ people
  - $n = 50$ people

```
sim_birthday(23)
```

```
[1] 0.5022
```

```
sim_birthday(50)
```

```
[1] 0.968
```

Results: - For $n = 23$: ≈ 0.507 (matches the analytical probability). - For $n = 50$: ≈ 0.970

Advantages of Simulations

1. **Flexibility**:
   - Works even when assumptions like uniformity or independence are adjusted.
2. **Verification**:
   - Confirms analytical results through experimental data.
3. **Real-World Application**:
   - Useful when exact formulas are unavailable or too complex.

- Simulations provide a practical approach to solving probability problems.

Practical Implications of the Birthday Problem

- Beyond recreational math, the birthday problem has real-world applications:
  - **Cryptography**:
    - Used to analyze the likelihood of hash collisions.
    - Essential in blockchain systems and digital signatures.
- These principles are crucial for understanding modern security systems.
- Explore the connection to cryptography further by reading the lecture notes at: https://martin-summer-1090.github.io/Probability_Introduction/

Summary

In this lecture, we have taken a first step towards some very basic probability notions and some basic steps in R. Isn't it amazing how much territory we could cover with so few concepts?

You have learned about how to think probabilistically about collision probabilities and how to solve for them analytically, by simulation, and for large numbers by approximation.

We have convinced ourselves using this knowledge only and taking on faith that the probability of independent events is the product of their individual probabilities, that the cryptographic Hash-function SHA-256, while it can produce collisions in theory, practically the number of hashes to make such a collision occur would be so large that we can be confident that hashing bit strings with SHA-256 gives us a unique fingerprint practically with certainty.

Key Concepts: Probability

1. **Random Experiment**: A process leading to an uncertain outcome.
2. **Sample Space**: The collection of all possible outcomes of a random experiment.
3. **Basic Outcome**: A possible outcome of a random experiment.
4. **Event**: A subset of basic outcomes. Any event that contains a single outcome is called a simple event.
5. **Classical Probability**: Defined by finding or making equally probable cases and then counting them.

$$P(A) = \frac{\text{Cases where } A \text{ occurs}}{\text{Total number of cases}}$$

6. **Relative Frequency Probability**: Defined as the number of times an event A occurs in repeated trials divided by the total number of trials.

Key Concepts: R

1. **Objects**: Arbitrary names that can store different values and data types.
2. **Functions**: R objects that can accept other R objects as arguments, operate on them, and return a new object.
3. **Scripts**: Files that store sequences of R commands, allowing them to be saved, reopened, and executed.
4. **Using Packages**: Extending R's functionality by importing libraries.
5. **Finding Help**: Accessing documentation to learn about R functions and packages.
6. **The Functions `sample` and `replicate`**: Tools for random sampling and repeating experiments.

Applications

1. **Constructing a Coin on the Computer and Tossing It an Arbitrary Number of Times**

   Simulating coin tosses to understand randomness.
2. **The Birthday Problem**

   Solving the problem analytically and through simulation.
3. **Extrapolating the Birthday Problem**

   Analyzing cryptographic collision resistance of hash-functions by applying ideas from probability.

Project 1:

Introduction: Real-World Importance of Unique Identifiers in Finance

- Modern financial systems assign a **unique identifier** to every transaction to ensure:
  - **Transparency**
  - **Accountability**
  - **Security**
- As transaction volumes grow, the risk of **identifier collisions** increases.

The Birthday Problem Analogy

- Collisions occur due to the combinatorial nature of the problem.
- Higher transaction volumes increase the likelihood of collisions.
- Implications of collisions:
  - **Processing errors**
  - **Fraudulent transactions**
  - **Misattribution of funds**

Hash Functions and Collision Prevention

- Financial systems often use **hash functions** (e.g., SHA-256) to prevent collisions.
- Properties of hash functions:
  - Map data to a large output space (e.g., 2^{256}).
  - **Extremely low probability of collision**, even for billions of transactions.

Project Overview

In this project, you will: 1. Simulate a system assigning transaction identifiers. 2. Investigate the probability of identifier collisions. 3. Discuss implications for system design and security.

Simulate Transaction Identifiers

- Write a function to simulate transaction identifiers:
  - Each identifier is randomly chosen from a pool of size M (e.g., $M = 10^6$ or $M = 10^9$).
- Simulate n transactions and check for duplicates using R's `duplicated()` function.

Compute Collision Probabilities

- Simulate n transactions multiple times.
- Estimate the collision probability as the fraction of simulations with at least one collision.
- Example calculations:
  - $n = 10^3, n = 10^6, n = 10^9$
  - $M = 10^6, M = 10^9$

Visualize the Results

- Plot the collision probability as a function of n for different values of M.
- Use R's plotting functions or explore R's help system or an LLM for guidance.
- Example goals:
  - Show trends for $M = 10^6$ and $M = 10^9$.
  - Compare the impact of transaction volume n.

Investigate System Design

- Reflect on how the size of the pool M influences collision risk.
- Explore what happens when transaction volume n increases significantly.
  - **Larger M reduces collision risk.**
  - **Higher n increases collision probability.**

Financial Implications

- Example Scenario:
  - $M = 10^6$ identifiers
  - Daily transaction volume $n = 10^5$
- Questions:
  - What is the collision risk?
  - What are the consequences of a collision?
    - Failed transactions
    - Fraud risks
  - How can financial systems mitigate these risks?
    - Increase M
    -