



# Beginning Mathematica and Wolfram for Data Science

Applications in Data Analysis, Machine  
Learning, and Neural Networks

—  
*Second Edition*

—  
Jalil Villalobos Alva

**Apress®**

# **Beginning Mathematica and Wolfram for Data Science**

**Applications in Data Analysis,  
Machine Learning,  
and Neural Networks**

**Second Edition**

**Jalil Villalobos Alva**

**Apress®**

# ***Beginning Mathematica and Wolfram for Data Science: Applications in Data Analysis, Machine Learning, and Neural Networks***

Jalil Villalobos Alva  
Mexico City, Mexico

ISBN-13 (pbk): 979-8-8688-0347-5  
<https://doi.org/10.1007/979-8-8688-0348-2>

ISBN-13 (electronic): 979-8-8688-0348-2

Copyright © 2024 by Jalil Villalobos Alva

This work is subject to copyright. All rights are reserved by the Publisher, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, reuse of illustrations, recitation, broadcasting, reproduction on microfilms or in any other physical way, and transmission or information storage and retrieval, electronic adaptation, computer software, or by similar or dissimilar methodology now known or hereafter developed.

Trademarked names, logos, and images may appear in this book. Rather than use a trademark symbol with every occurrence of a trademarked name, logo, or image we use the names, logos, and images only in an editorial fashion and to the benefit of the trademark owner, with no intention of infringement of the trademark.

The use in this publication of trade names, trademarks, service marks, and similar terms, even if they are not identified as such, is not to be taken as an expression of opinion as to whether or not they are subject to proprietary rights.

While the advice and information in this book are believed to be true and accurate at the date of publication, neither the authors nor the editors nor the publisher can accept any legal responsibility for any errors or omissions that may be made. The publisher makes no warranty, express or implied, with respect to the material contained herein.

Managing Director, Apress Media LLC: Welmoed Spahr  
Acquisitions Editor: Melissa Duffy  
Development Editor: James Markham  
Editorial Project Manager: Gryffin Winkler  
Copyeditor: Kim Burton

Cover designed by eStudioCalamar

Cover image designed by Mathew Schwartz on Unsplash ([www.unsplash.com](http://www.unsplash.com))

Distributed to the book trade worldwide by Springer Science+Business Media New York, 1 New York Plaza, Suite 4600, New York, NY 10004-1562, USA. Phone 1-800-SPRINGER, fax (201) 348-4505, e-mail [orders-ny@springer-sbm.com](mailto:orders-ny@springer-sbm.com), or visit [www.springeronline.com](http://www.springeronline.com). Apress Media, LLC is a California LLC and the sole member (owner) is Springer Science + Business Media Finance Inc (SSBM Finance Inc). SSBM Finance Inc is a **Delaware** corporation.

For information on translations, please e-mail [booktranslations@springernature.com](mailto:booktranslations@springernature.com); for reprint, paperback, or audio rights, please e-mail [bookpermissions@springernature.com](mailto:bookpermissions@springernature.com).

Apress titles may be purchased in bulk for academic, corporate, or promotional use. eBook versions and licenses are also available for most titles. For more information, reference our Print and eBook Bulk Sales web page at <http://www.apress.com/bulk-sales>.

Any source code or other supplementary material referenced by the author in this book is available to readers on GitHub. For more detailed information, please visit <https://www.apress.com/gp/services/source-code>.

If disposing of this product, please recycle the paper

*To my family, who supported me in all aspects*



# Table of Contents

About the Author ..... xiii

About the Technical Reviewer ..... xv

Acknowledgments ..... xvii

Introduction ..... xix

**Chapter 1: Introduction to Mathematica ..... 1**

    Why Mathematica? ..... 1

    The Wolfram Language ..... 2

    Structure of Mathematica ..... 3

        Design of Mathematica ..... 6

    Mathematica Environment ..... 9

        Notebook Interface ..... 9

        Text Processing ..... 12

        Palettes ..... 14

        Notebook Style and Features ..... 15

    Expression in Mathematica ..... 18

        Assigning Values ..... 19

        Built-in Functions ..... 22

        Dates and Time ..... 23

        Strings ..... 25

        Basic Plotting ..... 27

        Logical Operators and Infix Notation ..... 30

        Algebraic Expressions ..... 33

        Solving Algebraic Equations ..... 34

        Using Wolfram Alpha Inside Mathematica ..... 37

        Delayed and Immediate Expressions ..... 40

TABLE OF CONTENTS

Improving Code ..... 41

    Code Performance ..... 42

    Handling Errors ..... 43

    Debugging Techniques ..... 45

How Mathematica Works ..... 47

    How Computations are Made (Form of Input)..... 47

    Searching for Assistance ..... 50

    Notebook Security ..... 53

Summary..... 54

**Chapter 2: Data Manipulation ..... 55**

    Lists ..... 55

        Types of Numbers..... 56

        Working with Digits ..... 60

        A Few Mathematical Functions ..... 61

        Numeric Function ..... 63

    Lists of Objects ..... 64

        List Representation..... 65

        Generating Lists..... 65

        Arrays of Data..... 68

        Nested Lists..... 71

        Vectors..... 72

        Matrixes..... 73

        Matrix Operations ..... 75

        Restructuring a Matrix..... 76

    Manipulating Lists..... 77

        Retrieving Data ..... 77

        Assigning or Removing Values ..... 79

    Structuring List ..... 82

        Criteria Selection ..... 84

    Summary..... 87

<b>Chapter 3: Working with Data and Datasets</b>	<b>89</b>
Operations with Lists	89
Arithmetic Operations to a List	90
Applying Functions to a List	91
Defining Own Functions	93
Pure Functions	95
Indexed Tables	96
Tables with the Wolfram Language	96
Associations	101
Dataset Format	103
Constructing Datasets	103
Accessing Data in a Dataset	109
Adding Values	113
Dropping Values	117
Filtering Values	119
Applying Functions	122
Functions by Column or Row	127
Joining and Merging Datasets	132
Customizing a Dataset	134
Generalization of Hash Tables	138
Summary	145
<b>Chapter 4: Import and Export</b>	<b>147</b>
Importing Files	148
CSV and TSV Files	148
XLSX Files	150
JSON Files	153
Web Data	156
Semantic Import	157
Quantities	159
Datasets with Quantities	160
Costume Import (Dealing with Large Datasets)	164

TABLE OF CONTENTS

- Export..... 166
  - Other Formats..... 169
  - XLS and XLSX Formats ..... 173
  - JSON Formats..... 174
  - Content File Objects ..... 177
  - Searching Files with Wolfram Language ..... 178
- Connecting to External Services ..... 179
  - External Connections..... 179
  - External Resources..... 181
  - Database and File Operations (SQL) ..... 184
- Summary..... 186
- Chapter 5: Data Visualization ..... 187**
  - Basic Visualization ..... 187
    - 2D Plots ..... 187
    - Plotting Data ..... 191
    - Plotting Defined Functions ..... 196
  - Customizing Plots ..... 197
    - Adding Text to Charts..... 197
    - Frame and Grids ..... 200
    - Filled Plots ..... 202
    - Filling Patterns and Gradient..... 204
  - Combining Plots ..... 206
    - Multiple Plots..... 207
    - Multiaxis Plots ..... 210
    - Coloring Plot Grids..... 211
  - Colors Palette..... 216
  - 3D Plots..... 218
    - Customizing 3D Plots..... 219
    - Hue Color Function and List3D ..... 220

Contour Plots.....	222
3D Plots and 2D Projections .....	226
Plot Themes .....	227
Summary.....	231
<b>Chapter 6: Statistical Data Analysis .....</b>	<b>233</b>
Random Numbers .....	233
Random Sampling .....	235
Systematic Sampling.....	237
Commons Statistical Measures .....	239
Measures of Central Tendency .....	239
Measures of Dispersion.....	240
Statistical Charts.....	242
Bar Charts.....	242
Histograms .....	245
Pie Charts and Sector Charts.....	250
Box Plots.....	252
Distribution Chart .....	254
Charts Palette.....	256
Ordinary Least Squares Method.....	262
Pearson Coefficient .....	264
Linear Fit .....	265
Model Properties .....	266
Summary.....	269
<b>Chapter 7: Data Exploration.....</b>	<b>271</b>
Wolfram Data Repository .....	271
Wolfram Data Repository Website .....	272
Selecting a Category .....	274
Extracting Data from the Wolfram Data Repository .....	275
Accessing Data Inside Mathematica .....	278
Data Observation and Querying.....	281

TABLE OF CONTENTS

Descriptive Statistics .....	287
Table and Grid Formats .....	289
Dataset Visualization .....	293
Data Outside Dataset Format .....	296
2D and 3D Plots .....	297
Summary.....	301
<b>Chapter 8: Machine Learning with the Wolfram Language.....</b>	<b>303</b>
Gradient Descent Algorithm .....	303
Getting the Data.....	304
Algorithm Implementation .....	305
Multiple Alphas .....	307
Linear Regression .....	308
Predict Function .....	308
Boston Dataset .....	309
Model Creation .....	310
Model Measurements.....	316
Model Assessment .....	318
Retraining Model Hyperparameters.....	319
Logistic Regression.....	321
Titanic Dataset.....	321
Data Exploration .....	325
Classify Function .....	327
Testing the Model .....	332
Data Clustering .....	338
Clusters Identification.....	338
Choosing a Distance Function .....	340
Identifying Classes .....	342
K-Means Clustering.....	343
Dimensionality Reduction.....	345
Applying K-Means .....	347
Changing the Distance Function.....	349



Different k's .....	350
Cluster Classify .....	353
Summary.....	357
<b>Chapter 9: Neural Networks with the Wolfram Language .....</b>	<b>359</b>
Layers .....	360
Input Data .....	360
Linear Layer.....	360
Weights and Biases .....	361
Initializing a Layer .....	362
Retrieving Data .....	364
Mean Squared Layer .....	365
Activation Functions .....	368
Softmax Layer .....	372
Function Layer.....	374
Encoder and Decoders.....	375
Encoder .....	375
Pooling Layer.....	379
Decoders .....	380
Applying Encoder and Decoders.....	382
NetChains and Graphs .....	383
Containers .....	383
Multiple Chains.....	386
NetGraphs.....	387
Combining Containers .....	393
Network Properties.....	395
Exporting and Importing a Model .....	399
Summary.....	402
<b>Chapter 10: Neural Networks Framework .....</b>	<b>403</b>
Training a Neural Network .....	403
Data Input.....	403
Training Phase .....	405

TABLE OF CONTENTS

Model Implementation..... 406

Batch Size and Rounds..... 408

Training Method (NetTrain) ..... 412

Measuring Performance ..... 413

Model Assessment ..... 415

Exporting a Neural Network ..... 417

Wolfram Neural Net Repository..... 418

    Selecting a Neural Net Model..... 419

    Accessing Inside Mathematica..... 421

    Retrieving Relevant Information ..... 422

LeNet Neural Network..... 423

    LeNet Model ..... 423

    MINST Dataset..... 424

    LeNet Architecture..... 425

    MXNet Framework..... 426

    Preparing LeNet..... 428

    LeNet Training..... 430

    LeNet Model Assesment..... 432

    Testing LeNet..... 434

GPT and LLM Basics..... 436

    A Brief Overview ..... 437

    LLM in the Wolfram Language..... 437

    Chat Notebooks ..... 438

    Wolfram Prompt Repository ..... 441

    LLM Functionalities ..... 444

    GTP-1 and GPT-2 Models..... 446

Final Remarks ..... 448

**Index..... 451**

# About the Author

**Jalil Villalobos Alva** is a Wolfram Language programmer and Mathematica user. He graduated with a degree in engineering physics from the Universidad Iberoamericana in Mexico City. His research background comprises quantum physics, bioinformatics, proteomics, and protein design. His academic interests cover the topics of quantum technology, bioinformatics, machine learning, artificial intelligence, stochastic processes, and space engineering. During his idle hours, he likes playing soccer, swimming, and listening to music.

# About the Technical Reviewer



**Andrew Yule** is a co-founder and managing partner of Pontem Analytics, a global consulting company in the energy industry specializing in combining domain expertise with data-driven solutions. Andrew has over 13 years of professional experience leveraging the Wolfram Language and was the recipient of the Wolfram Innovator Award in 2017. He is an editor for the Society of Petroleum Engineers, The Way Ahead magazine, and is also currently a member of the Young Entrepreneurial Council. His technical background includes a bachelor's degree in chemical engineering from the Colorado School of Mines and a master's degree in data science from Southern Methodist University.

# Acknowledgments

I want to thank the collective support and guidance received throughout the development of this project's second edition. The contributions of numerous past and present individuals have played an integral role in shaping this work. Their assistance, feedback, and mentorship have been invaluable, enriching the content and presentation of this edition. I also want to thank the technical and staff reviewers for their valuable comments and feedback during this manuscript. They both helped me improve the material's presentation and theoretical work. And finally, I would like to thank Las Des Nestor and "Los Betos" for teaching me great mastery.

# Introduction

Welcome to *Beginning Mathematica and Wolfram for Data Science*.

Why is data science important nowadays? Data science is an active topic that is evolving daily; new methods, techniques, and data are created daily. Data science is an interdisciplinary field involving scientific methods, algorithms, and systematic procedures to extract data sets and thus better understand the data in its different structures. It is a continuation of some theoretical data analysis fields such as statistics, data mining, machine learning, and pattern analysis. With a unique objective, to extract quantitative and qualitative information of value from the data being recollected from various sources, and thus be able to objectively count an event for decision-making, product development, pattern detection, or identification of new business areas.

## Data Science Roadmap

Data science carries out a series of processes to solve a problem, which includes data acquisition, data processing, model construction, communication of results, and data monitoring or model improvement. The first step is to formalize an objective in the investigation. From the object of the investigation, you can proceed to the data acquisition sources. This step focuses on finding the right data sources. The product of this path is usually raw data, which must be processed before it can be handled. Data processing includes transforming the data from a raw form to a state in which it can be reproduced to construct a mathematical model. Proceeding to the construction of the model, a stage that intends to obtain the information by making predictions in accordance with the conditions established in the early stages. Here, the appropriate techniques and tools, which consist of different disciplines, are used. The objective is to obtain a model that provides the best results. The next step is to present the outcome of the study. Which consists of reporting the results obtained and whether they are congruent with the established research objective. Finally, it comes to data monitoring, with the intention of keeping the data updated because data can change constantly and in different ways.



# Data Science Techniques

Data science includes analysis techniques from different disciplines, such as mathematics, statistics, computer science, and numerical analysis. The following are some disciplines and techniques used.

- Statistics (linear, multiple regressions, least squares method, hypothesis testing, analysis of variance (ANOVA), cross-validation, resampling methods)
- Graph theory (network analysis, social network analysis)
- Artificial intelligence
- Machine learning
- Supervised learning (natural language processing, decision trees, naive bayes, nearest neighbors. support vector machine)
- Unsupervised learning (cluster analysis, anomaly detection, K-means cluster)
- Deep learning (artificial neural networks, deep neural networks)
- Stochastic processes (Monte Carlo methods, Markov chains, time series analysis, nonlinear models)

Even though many techniques exist, this list only shows a part of it since research on data science, machine learning, and artificial neural networks is constantly increasing.

## Prerequisites

This book is intended for readers who want to learn about Mathematica / Wolfram Language and implement it in data science; it focuses on the basic principles of data science as well as for programmers outside of software development, that is, people who write code for their academic and research projects, including students, researchers, teachers, and many others. The general audience is not expected to be familiar with Wolfram Language or with the front-end program Mathematica, but little or any experience is welcome. Previous knowledge of the syntax would be an advantage in

understanding how the commands work in Mathematica. If this is not the case, the book provides the basic concepts of the Wolfram Language syntax. The fundamental structure of expressions in the Wolfram Language. Basic handling and understanding of Mathematica notebooks.

Prior knowledge or some experience with programming, mathematical concepts such as numbers, trigonometric functions, and basic statistics are useful, along with some understanding of mathematical modeling, which is also helpful but not compulsory.

Wolfram Language is different from many other languages but very intuitive and user-friendly to learn.

The book aims to teach the general structure of the Wolfram Language, data structures, objects, and rules for writing efficient code, and at the same time, teach data management techniques that allow them to solve problems in a simple and effective way. Provide the reader with the basic tools of the Wolfram Language, such as creating structured data, to support the construction of future practical projects.

For this new version, all the programming was carried out on a MacBook Air M1 with Sonoma 14 environment with the installation of version 13.3.1.0 and 14 of Wolfram Mathematica. Wolfram Mathematica is currently supported in other environments such as Linux, Windows, and macOS. The code found in the book works with both the Pro and Student versions.

## Book Conventions

Throughout the book, you may come across different words written distinctly from others. Throughout the book, the words command, built-in functions, and functions may be used as synonyms that mean Wolfram Language commands written in Mathematica. So, a function will be written in the form of the real name; for example, RandomInteger.

The evaluation of expressions appears in the Mathematica In/Out format; the same applies to blocks of code.

```
In[#]:= "Hello World!"
```

```
Out[#]= "Hello World!"
```

# The Layout

The book is written in a compact and focused way to cover the basic ideas behind the Wolfram Language and cover details on more complex topics. Some chapters have been revised and redesigned in this new version to focus on novice and advanced topics.

Chapter 1 discusses the starting topics of the Wolfram Language, basic syntax, and basic concepts with some example application areas, followed by an overview of the basic operations and debugging techniques, and concludes by discussing security measures within a Mathematica session.

Chapter 2 provides the key concepts and commands for data manipulation, sampling, types of objects, and some concepts of linear algebra—the introduction to lists, an important concept to understand in the Wolfram Language.

Chapter 3 discusses how to work properly with data and the initiation of the core structures for creating a dataset object, introducing concepts like associations and association rules are discussed with a conclusion remarking how associations and dataset constructions can be interpreted as a generalization of a hash table aiming to expose a better understanding of internal structures inside the Wolfram Language, including an overview of performing operations on a list and between lists and then discussing various techniques applied to dataset objects.

Chapter 4 exposes the main ideas behind importing and exporting data with examples throughout the chapter with common and newly added file formats. It also presents a very powerful command known as `SemanticImport`, which can import data elements that are natural language.

Chapter 5 covers the topic areas for new data visualization, common data plots, data colors, data markers, and how to customize a plot. Basic commands for 2D plots and 3D plots are presented, too.

Chapter 6 introduces the statistical data analysis. Starting with random data generation begins by introducing some standard statistical measures, followed by a discussion on creating statistical charts and performing an ordinary least square method.

Chapter 7 exposes the basis for data exploration and reviews a central discussion on the Wolfram Data Repository. Performing descriptive statistics and data visualization inside Fisher's Irises dataset objects is also covered.

Chapter 8 starts with machine learning concepts and techniques, such as gradient descent, linear regression, logistic regression, and cluster analysis, including examples from various datasets like the Boston and Titanic datasets and newly implemented features.

Chapter 9 introduces the key ideas and the basic theory to understand the construction of neural networks in the Wolfram Language, such as layers, containers, and graphs. The MXNet framework in the Wolfram Language scheme is also discussed.

Chapter 10 concludes the book by discussing training neural networks in the Wolfram Language. In addition, the Wolfram Neural Net Repository is discussed with an example application, examining how to access data inside Mathematica and the retrieval of information, such as credit risk modeling fraud detection, and concluding with the example of the LeNet neural network, reviewing the idea behind this neural network and exposing the main points on the architecture with the help of the MXNet graph operations and a final road map on the creation, evaluation, and deployment of predictive models with the Wolfram Language. In this new version, LLM (large language model) features are introduced with the connection to GPT services, use of chat cells, and presentation of the GPT-1 and GPT-2 models.

## CHAPTER 1

# Introduction to Mathematica

The chapter begins with a preliminary introduction to why Mathematica is a useful and practical tool. It explores the core concepts of the Wolfram Language and its syntax. It starts by explaining the internal structure of Mathematica and how to add code effectively. The concept of a notebook is introduced, which is important to understand the type of format that Mathematica handles. The chapter examines this interface class and demonstrates how notebooks simultaneously support code and text. In this way, a notebook is a computable text file. Next, you inspect various add-ons that can be employed within a notebook to help the user maximize their code's capabilities.

The next section demonstrates how to write expressions in Mathematica, examining topics such as arithmetic, algebra, symbols, global and local variables, built-in functions, date and time formats, plotting functions, logical operators, performance measures, delayed expressions, and accessing Wolfram Alpha. You then look at how Mathematica performs code computations, including its accepted varieties of inputs and the evaluation of these inputs. This chapter concludes with tips for seeking support within Mathematica, managing and handling errors, searching for solutions, and safely dealing with security concerns in notebooks that incorporate dynamic content.

## Why Mathematica?

Mathematica is a mathematical software package created by Stephen Wolfram more than 35 years ago. Its first official version (Mathematica 1.0) emerged in 1988 and was created as an algebraic computational system capable of handling symbolic computations. However, Mathematica has established itself as a tool capable of performing complex tasks efficiently, automatically, and intuitively. Mathematica is

widely used in many disciplines like engineering, optics, physics, graph theory, financial engineering, game development, and software development.

Mathematica provides a complete, integrated platform to import, analyze, and visualize data. Mathematica does not require plug-ins. It also has a mixed syntax, performing both symbolic and numerical calculations. It provides an accessible way to read the code with the implementation of notebooks as a standard format, which also serves to create detailed reports of the processes carried out. Mathematica can be characterized as a powerful platform enabling efficient and concise forms of work. Among computer languages, the Wolfram Language falls into the group of programming languages classified as a high-level, multi-paradigm interpreted language. Unlike conventional programming languages, the Wolfram Language adheres to unique rules, facilitating order and clear, compact code composition.

## The Wolfram Language

Mathematica is powered by the Wolfram Language, an interpreted high-level programming language that covers both symbolic and numeric capabilities. To understand the Wolfram Language, it is necessary to remember that the language's core nature resembles a normal mathematical text, as opposed to other programming languages' syntax. The following describes some remarkable features of the Wolfram Language.

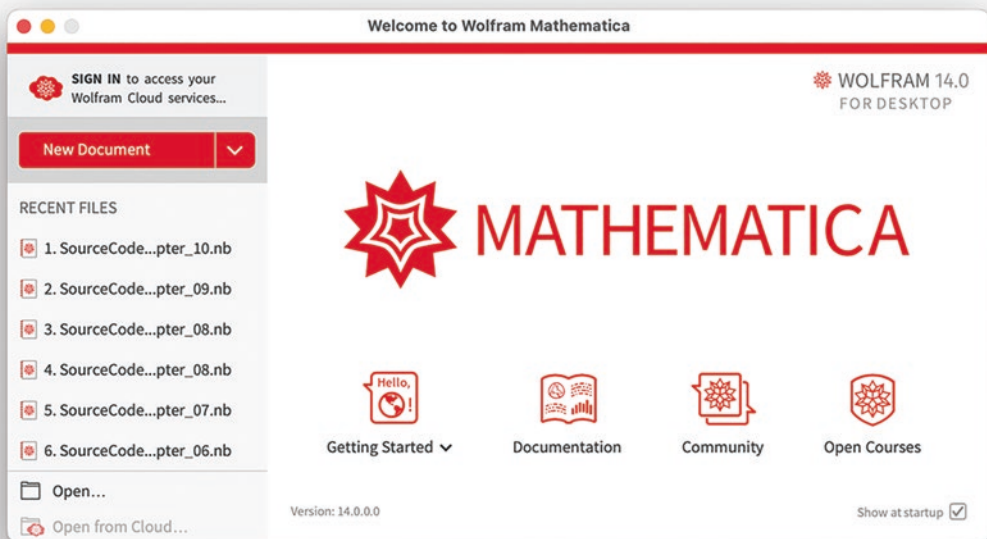
- The first letter of a built-in function word is uppercase and is also human-readable.
- Any element introduced in the language is taken as an expression.
- Expressions take values consisting of the Wolfram Language atomic expressions.
  - A symbol made up of letters, numbers, or alphanumeric contents
  - Four types of numbers: integers, rational, real, and complex
  - The default character string is written within the quotation marks ( " ")



- In Mathematica, there are three ways to group expressions.
  - Parentheses group terms within an expression  $(\text{expr1} + \text{expr2}) + (\text{expr3})$ .
  - Command entries are enclosed by brackets `[ ]`. Also, square brackets enclose the arguments of a built-in function, `F[x]`.
  - Mathematica uses curly braces `{ }` (e.g., `{a, b, c}`) to represent lists, arrays, matrixes, and other collections.

## Structure of Mathematica

Before entering code, you need to get the layout of Mathematica. To launch Mathematica, go to your Applications folder and select the Mathematica icon. This action brings up the new welcome screen, illustrated in Figure 1-1.



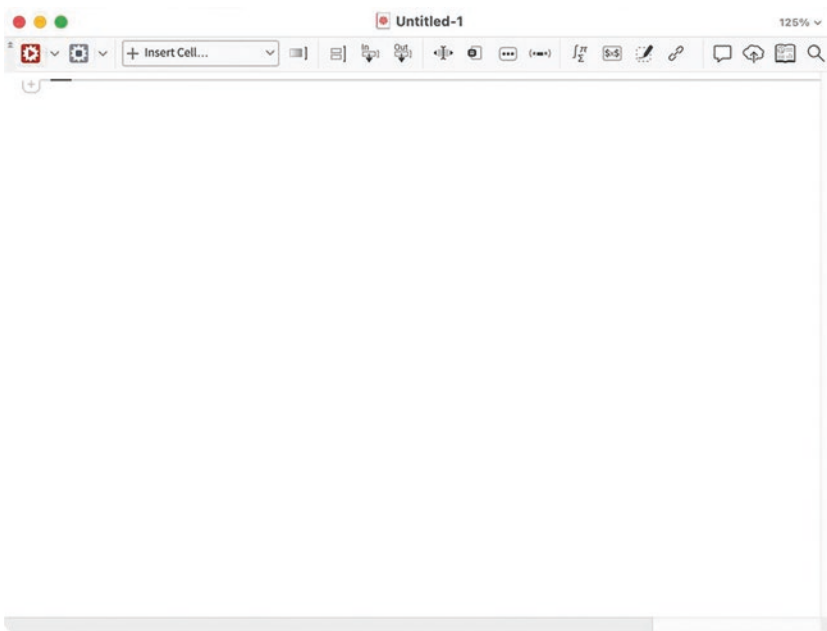
**Figure 1-1.** The default welcome screen for Mathematica's latest version

---

**Tip** The startup window offers valuable information for new and adept users, including the Mathematica version, access to documentation, resources, and the Wolfram community, among other things.

---

After the startup screen appears, you can create a new notebook by selecting the New Document button, and a blank page should appear like the one shown in Figure 1-2. New documents can also be created by selecting File ► New ► Notebook or with the ⌘+N (macOS) or Ctrl+N (Win) keyboard shortcut command.



**Figure 1-2.** A blank notebook ready to receive input

The blank document that appears is called a notebook, and it's the core interaction between the user and Mathematica. Notebooks can be saved locally from the menu bar by selecting File ► Save (or Save as). Initializing Mathematica always exhibits an untitled notebook. Notebooks serve as the standard document format. They can be customized to display text alongside computations. However, the key feature of Mathematica lies in its capacity to perform computations, extending beyond numerical calculations, regardless of the notebook's purpose.

---

**Note** Mathematica version 13.1 introduced a new default assistant toolbar.

---

Mathematica's notebooks are separated into input spaces called cells. Cells are represented by the square brackets on the notebook's right side. Each input and output cell has its bracket. Brackets enclosed by larger brackets are related computations, whether input or output. Grouped cells are represented by nested brackets that contain the whole evaluation cell. Other cells can be grouped by selecting and grouping them with the right-click option. Cells can also have the capability to show or hide input by simply double-clicking the cells. To add a new cell, move the text cursor down, and a flat line should appear, marking the new cell ready to receive input expressions. The plus tab in the line is the assistant input tab, showing the various types of input supported by Mathematica. Figure 1-3 displays grouped input (In[-]) and output (Out[-]) cells.

```
In[ ]:= "Hello World" ]
Out[ ]= Hello World ]
```

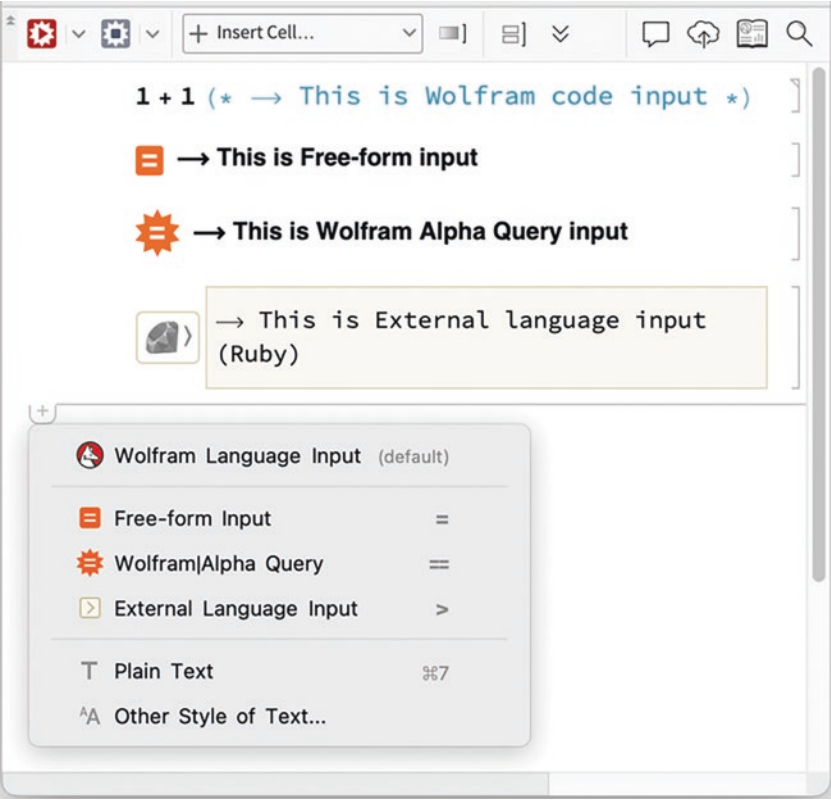
**Figure 1-3.** Expression cells are grouped by input and output

There are four main input types. The default input is the Wolfram Language code input. Free-form input is involved with Wolfram knowledge-base servers, and the results are shown in Wolfram Language syntax. Wolfram Alpha query is associated with results explicitly shown on the Wolfram Alpha website. External Language Input is built-in support for common external programming supported by Mathematica.

There are four main input types.

- Default input: Wolfram Language code input
- Free-form input: involved with Wolfram knowledge-base servers and the results are shown in Wolfram Language syntax
- Wolfram Alpha query: associated with results explicitly shown on the Wolfram Alpha website
- External language input: built-in support for common external programming supported by Mathematica

These are illustrated in Figure 1-4.



**Figure 1-4.** Main input types in Mathematica

---

**Tip** Keyboard shortcuts for front-end instruction commands are shown on the right or left side of each panel.

---

## Design of Mathematica

Now that you have the lay of the land of Mathematica’s basic format, you can learn the internal structure of how Mathematica works. Inside Mathematica, there are two fundamental processes: the Mathematica kernel and the graphical interface. The Mathematica kernel is the one that takes care of performing the programming computations; it is where the Wolfram Language is interpreted and is associated with each Mathematica session. The Mathematica interface allows the user to interact with the Wolfram Language functions and, at the same time, document your progress.

Each notebook contains cells, where the commands that the Mathematica kernel receives are written and then evaluated. Each cell has an associated number. There are two types of cells: the Input cell and the Output cell. These are associated with each other and have the following expressions: In[n]:= Expression and Out [n]: = Result or (“new expr”). The evaluations are listed according to which cell is evaluated first and continue in ascending order. When quitting the kernel session, all the information, computations made, and stored variables are relinquished, and the kernel is restarted, including the cell expressions. To quit a kernel session, select Evaluation ► Quit Kernel ► Local.

---

**Tip** To start a new kernel session, click Evaluation ► Start Kernel ► Local.

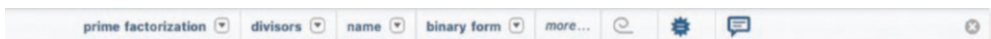
---

To begin, try typing the following computation.

```
In[1] := (11*17) + (4/2)
Out[1] = 189
```

The computation shows that In and Out have a number enclosed. This number is the number associated with the evaluated expression.

A suggestion bar appears after every expression is evaluated (see Figure 1-5). The suggestion bar in Mathematica is always visible unless the user hides it. But the suggestion bar offers suggestions for possible new commands or functions to be applied to the generated output. The suggestion bar can sometimes be helpful if you are unsure what to code next; if used wisely, it might be helpful.



**Figure 1-5.** *Suggestion bar for more possible evaluations*

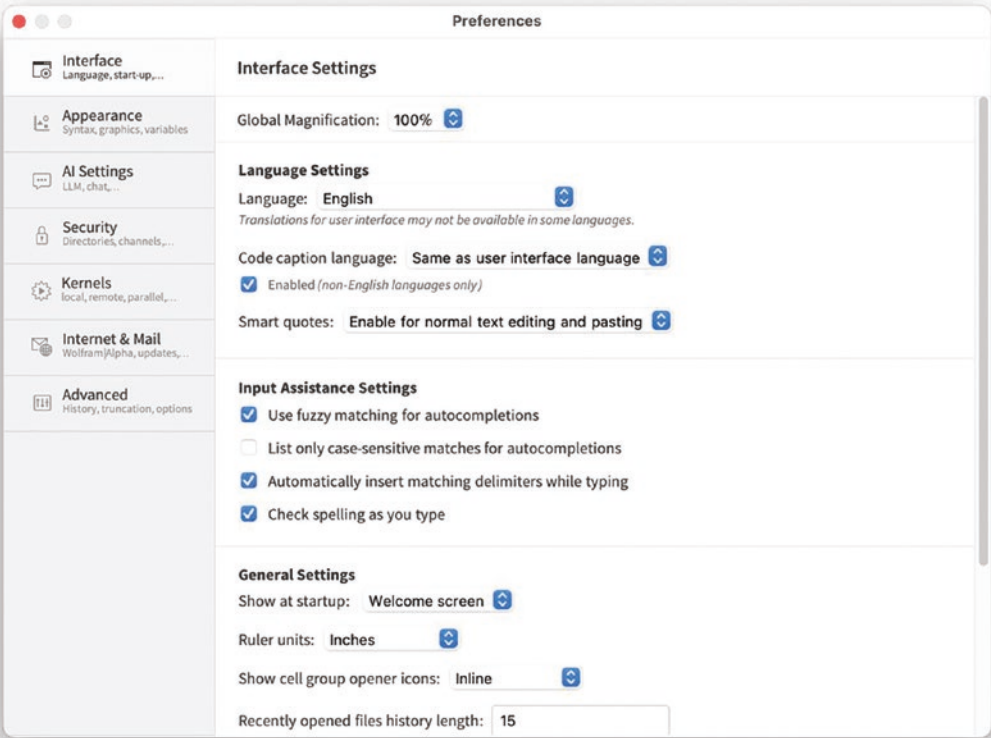
The input form of Mathematica is intuitive; to write in a Mathematica notebook, you just have to put the cursor in a blank space, and the cursor indicates that you are inside a cell that has not been evaluated. To evaluate a cell, click the keys [Shift + Enter], instructing Mathematica kernel to evaluate the expression written. The next chapter looks at the new form to evaluate expressions using the new toolbar.

To evaluate the whole notebook, go to the Evaluation tab on the toolbar and select Evaluate Notebook. If the execution of calculations takes more time than expected, you make a wrong execution of code, or if you want to seize a computation, Mathematica provides several ways to stop calculations. To abort a computation, go to

Evaluation ► Abort Evaluation. Alternatively, use the keyboard shortcut in Windows [Alt + .] or macOS [⌘ + .].

When a new notebook is created, the default settings are applied to every cell (input style). Nevertheless, preferences can be edited in Mathematica with various options. To access them, go to Edit ► Preferences. On macOS, the Preferences (settings) menu is located in the application menu, go to Mathematica ► Settings.

Once opened, a pop-up window appears (see Figure 1-6) with multiple tabs (Interface, Appearance, AI Settings, etc.). Basic customizations involve magnification, language settings, and other general instructions. The Appearance tab is related to code syntax color (i.e., symbols, strings, comments, errors, etc.). The AI Settings tab is the new tab associated with the LLM (large language model) evaluator. Other options belong to advanced settings that are not used in this book. Feel free to navigate each option.



**Figure 1-6.** Preferences window



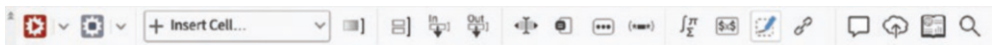
Later, you learn about in-depth settings and customization options for the notebook interface that allow you to tailor preferences.

## Mathematica Environment

This section explores the user interface of Mathematica, with a focus on the notebook interface, as well as the other user experience functionalities.

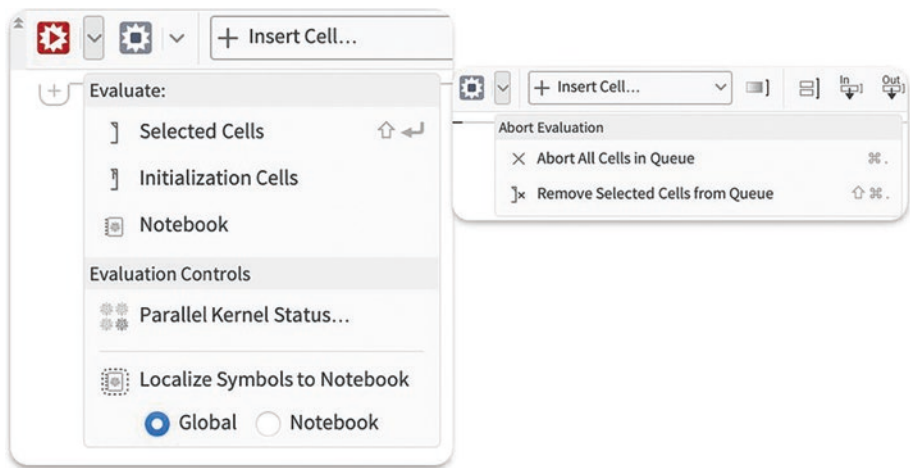
### Notebook Interface

Mathematica is always on the quest to improve user experience and boost productivity. In version 13.1, a big enhancement has been introduced—the seamless integration of a default toolbar (see Figure 1-7) across all standard notebook user interfaces (UIs).



**Figure 1-7.** The new UI default toolbar showcases essential tools and functionalities for efficient code development. Toolbar icons may vary by Mathematica version

This new toolbar (described left to right) includes several new features to enhance user experience. Evaluate allows users basic and costume code evaluation. Abort lets users cancel queued cells and remove chosen ones; both options are shown in Figure 1-8. These features can also be accessed via the keyboard, as previously mentioned.



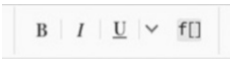
**Figure 1-8.** Extensive options for code evaluation and abort options in a notebook interface; the double arrow-like shape hides the toolbar

The other options integrate text cell formatting, offering styling options like cell style (title, subtitle, etc.) and cell color background. Users also benefit from the convenient cell management functions, such as grouping, dividing, merging cells, and inserting input/output of cells, all reduced to simple buttons. Continuing to options like extend selection, convert natural language into Wolfram Language code, collapse cells, insert comment, math form input, and LATEX rendering, users also have access to drawing canvas and hyperlink features. Finally, the rightmost section of the toolbar includes buttons for chat notebooks (utilizing LLM features), saving or publishing to the Wolfram Cloud, accessing documentation (local or web-based), and searching within the notebook.

---

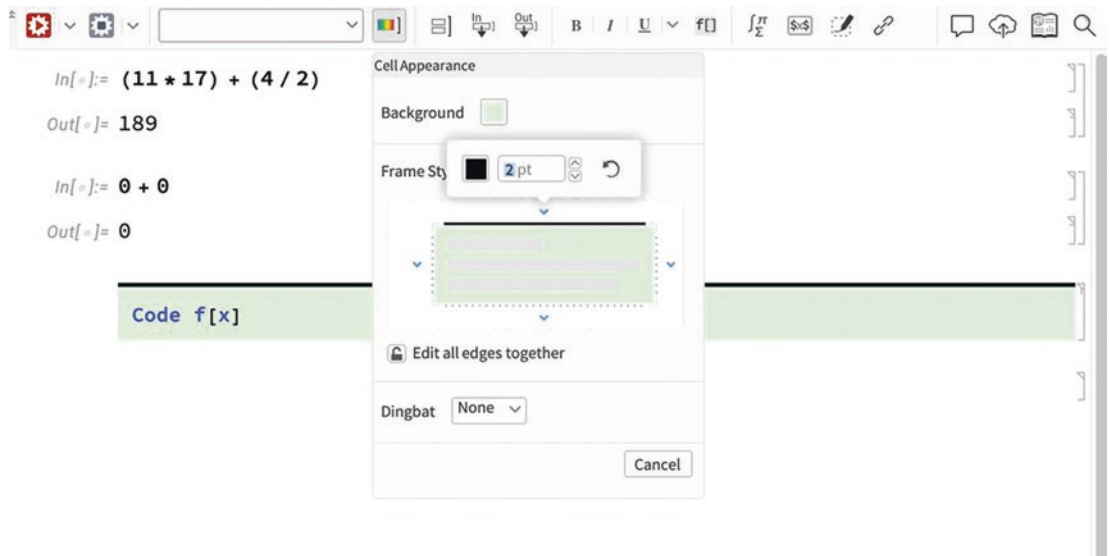
**Note** Different buttons may appear based on the selected cell type where the cursor resides, ranging from text to code formatting. Figure 1-7 shows for Wolfram Language code input cell. Figure 1-9 shows for text display cell.

---



**Figure 1-9.** Text cell options for bold, italic, and underline and insert code text evaluation and abort options in a notebook interface

This essential addition provides a coherent user experience and fosters a more streamlined, productive programming environment within Mathematica. For example, Figure 1-10 shows a code input cell with a colored background.



**Figure 1-10.** Light-green code input cell, with a 2pt black top margin

Besides the default toolbar, more improvements were made to the other toolbars, Ruler, Formatting, Templating, and Testing, as Figure 1-11 shows. The last two are not shown since they are more associated with a specific type of programmatic notebook, which is beyond the scope of this book.



**Figure 1-11.** Notebook application toolbar menu showcasing three distinct toolbars: formatting (upper), default (central), and ruler (lower)

---

**Note** To show or hide any toolbar, go to Window ► Toolbar. Toolbar availability varies by version.

---

The prominent toolbar Ruler indicates and adjusts the text margins of specified cells using draggable marks, offering control over the text format. The Formatting toolbar brings advanced textual design options, while the Templating and Testing toolbars (not shown in the image) facilitate the efficient creation of new templates and testing programmatic notebooks.

# Text Processing

Notebooks can include explanatory text, titles, sections, and subsections. The Mathematica notebook resembles a computable document rather than a programming command line. Text is useful for describing code and can be inserted into cells as text cells, which often relate to the corresponding computations. Mathematica allows you to work with multiple forms of text cells, including lines of text, chapters, formulas, items, bullets, and more. Like a word-processing tool, notebooks can have titles, chapters, sections, and subsections. By selecting Format ► Style, additional options become available. For more control over style cells, use the formatting toolbar (see Figure 1-12) found by navigating to Window ► Toolbar Formatting in the menu bar. The formatting toolbar streamlines cell styling, allowing users to justify text left, center, right, or fully.



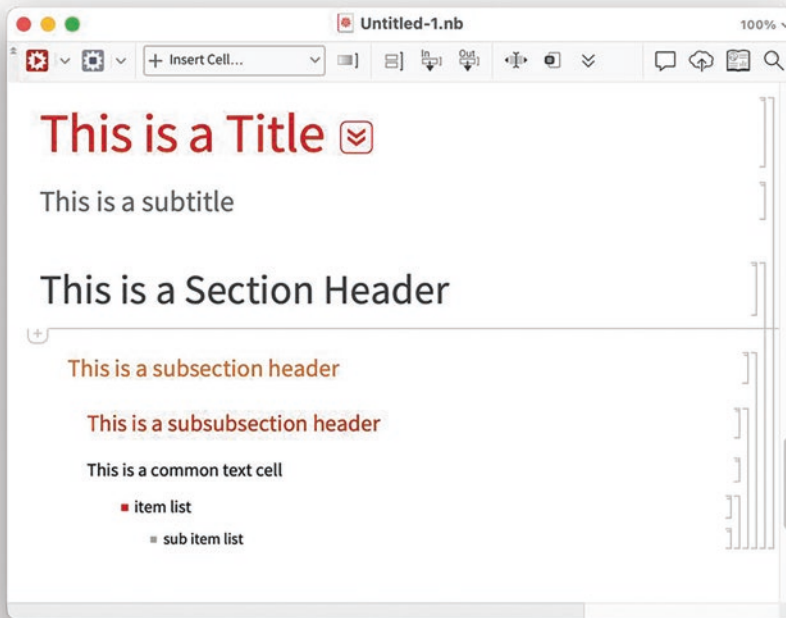
**Figure 1-12.** *The Style Format toolbar has a user-friendly interface for customizing text appearance*

The cell types can be arranged in different forms, depending on the notebook’s format. There are numerous forms to add text in a cell; the most straightforward is to type the text in the input cell, and the Assistant tab input automatically suggests converting it to text. Another alternative is to choose the cell type from the toolbars, with the input chooser or the shortcut (⌘+7 or Ctrl+7).

Styled text can be created with the formatting toolbar or by selecting the desired style in Format ► Style ► (title, chapter, text, code, input, etc.). In the Style menu, note the keyboard shortcuts for all the available text styles. It can be used instead of going into the menu bar every time. Plain text can also be converted into input text by formatting the cell in the Input style. There is no restriction in converting text; text can be converted into whatever style is supported in the format menu.

**Note** To convert text, highlight the text or select the cell that contains the text.

As shown in Figure 1-13, styled cells look different from others. Each style has a unique order by which a notebook is organized into cell groups. A title cell has a higher order in a notebook, so the other cells are anchored to the title cell, as shown in Figure 1-13, but it does not mean that if another title cell is added, both titles are grouped. If the title cell is collapsed, the title is the only displayed text.



**Figure 1-13.** A notebook with different format styles; this includes title, subtitle, section, subsections, plain text, item list, and subitem list

Text can be given a particular style, changed, and different formats applied throughout the notebook. By selecting Font or Show Fonts (macOS users) from the Format menu, a pop-up window appears, allowing you to change the font, font style, size, and other characteristics.

**Tip** To clear the format style of a cell, select the cell and then the right-click button and choose Clear Formatting.

---

## Palettes

Palettes show different ways to enter various commands into Mathematica. A diverse quantity of special characters and typesetting symbols are used in the Wolfram Language, which can be typed within expressions to more closely resemble mathematical text. The best way to access these symbols is by using the pallets built into Mathematica. To select a simple pallet, go to Pallets ► Basic Math Assistant. Each pallet has different tabs that stand for different categories with distinct commands and a variety of characters or placeholders that can be inserted using the pallets. To enter the symbol, type **ESC** followed by the name of the symbol, then ESC again. Try typing (**ESC a ESC**) to type the lowercase alpha Greek letter. Figure 1-14 shows the basic math assistant pallet in Mathematica.



**Figure 1-14.** The Basic Math Assistant palette

---

**Note** Hovering the mouse cursor over a symbol or character, an information tip pops up, showing the keyboard shortcut. This also applies to placeholders.

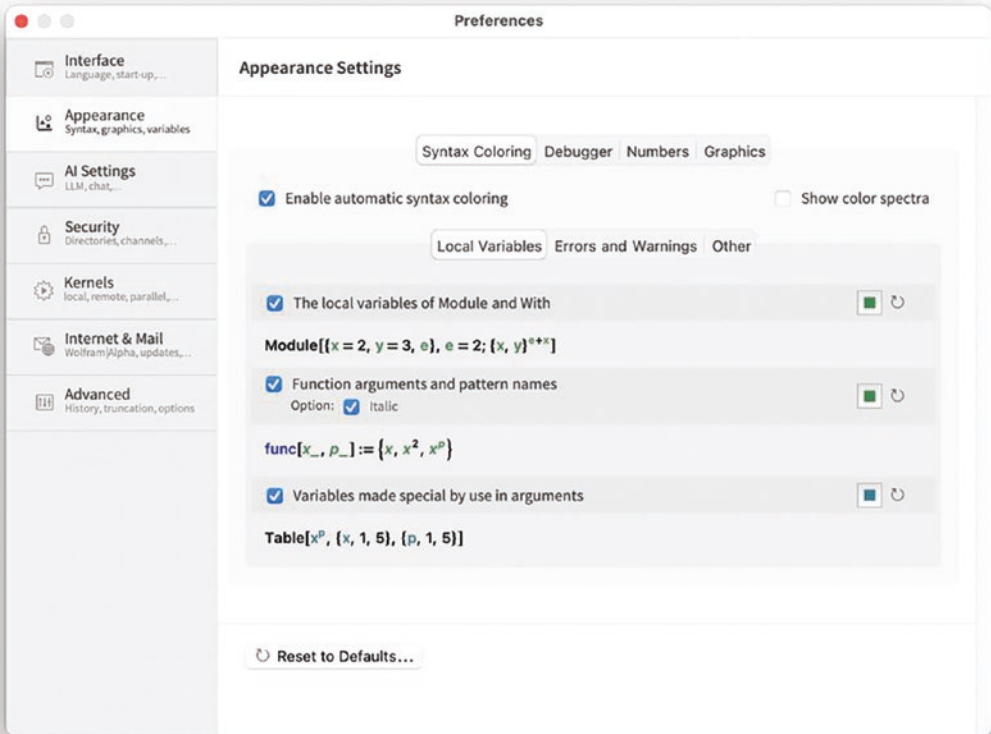
---

## Notebook Style and Features

In the new versions comprising version 13.0 and beyond, Mathematica has been refining and polishing its notebook interface by adding new features for a smoother user experience. One considerable enhancement involves the handling of extensive outputs within the user interface. Users can efficiently manage and interpret sizable outputs without overwhelming the notebook display or causing memory issues. The following







**Figure 1-17.** Notebook settings customization

Figure 1-17 shows Appearance Settings window. The Syntax Coloring tab is related to the visual representation of code elements (variables, errors, automatic coloring, highlighting, etc.). The Debugger tab includes coloring options about debugger highlights, breakpoints, and evaluation points. The Numbers tab offers multiple choices based on formatting and configuration choices, a few mentions are digits control numerical notation, among others. The Graphics tab allows you to choose the render of 2D and 3D graphics, from lowest to highest quality.

---

**Note** To change the colors of the code syntax options in the Appearance windows. Select the variables checkbox and click the green square. A color wheel pops up, allowing you to change the color. This process is the same for the three code setting options. Also, as you can see, there are different tabs.

---

## Expression in Mathematica

Basic arithmetic operations can be performed in Mathematica with a common, intuitive form.

```
In[3]:= (3*3) + (4/2)
Out[3] = 11
```

Mathematica also provides the capability to use a traditional mathematical notation. To insert a placeholder in the form, click [Ctrl + 6]. To indicate the product operation, use a space between expressions or add an asterisk (\*) between.

```
In[4]:= 1002 * 10
Out[4]= 100000
```

```
In[5]:= 2 1
Out[5]= 2
```

The standard Mathematica format aims to deliver the value closest to its regular form, so when dealing with decimal numbers or general math notation, Mathematica always gives you the best precision (involving, in some circumstances, infinite precision). However, it allows you to manipulate expressions numerically, to display numeric values, you use the N function. To insert the square root, type [Ctrl + 2].

```
In[6]:= 1/2 +  $\sqrt{2}$ 
Out[6]= 1/2 +  $\sqrt{2}$ 

In[7]:= N[1/2 +  $\sqrt{2}$ ]
Out[7]= 1.91421
```

You can manage the number precision of a numeric expression. In this case, you establish 10 decimal places.

```
In[8]:= N[77/13, 10]
Out[8]= 5.923076923
```

For a shortcut to work with the decimal point, just type a dot (.) anywhere in the expression, and with this, you are telling Mathematica to calculate the value with machine precision.

```
In[9]:= 4. + 2/13
Out[9]= 2.15385
```

Mathematica performs the sequence of operations from left to right, in line with the written expression, while adhering to the standard order of mathematical operations. To evaluate an expression without showing the result, you add a semicolon (;) after the end of the first term. In the following example, the 11/7 is evaluated but not shown, and the other term is displayed.

```
In[10]:= 11/7; Sqrt[4]
Out[10]= 2
```

The last form of code is called a compound expression. Expressions can be written in a single line of code, and with compound expressions, they are evaluated in the intended sequence. If you write the semicolon in each expression, Mathematica does not return the values, but they are evaluated.

```
In[11]:= 3*4; 100*100; Sqrt[4];Power[2,2];
Out[11]=
```

There is no output, but all the expressions have been evaluated. Later, you use compound expressions to understand the concept of delayed expressions. This basic feature of the Wolfram Language makes it possible for expressions to be evaluated but not displayed to save memory.

## Assigning Values

In the Wolfram Language, each variable requires a unique identifier that distinguishes it from the others. A variable in the Wolfram Language can be a union of more than one letter and digits; it must also not coincide with protected words—reserved words that refer to commands or built-in functions. Keep in mind that the Wolfram Language is case-sensitive. User variables are advised to be lowercase to avoid confusion with built-in symbols.

---

**Note** Mathematica supports assigning values to variables, which enables the effective handling of algebraic variables.

---

Undefined variables or symbols appear in blue font, while defined or recognized built-in functions appear in black. It is also true that the previously mentioned characteristics can be changed in the preferences window.

Use the keyboard shortcut Esc pi Esc (pi number) to write special constants and Greek letters. A symbol of a vertical ellipsis (:) should appear every time Esc is typed. Another choice is to write the first letter of the name, and a sub-menu showing a list of options should appear.

```
In[12]:= a=Pi
x=11
z+y
Out[12]=  $\pi$ 
Out[13]= 11
Out[14]=  $y+z$ 
```

In the previous example, Mathematica expresses each output with its cell, even though the input cell is just one. That is because Mathematica gives each cell a unique identifier. To access previous evaluations, the symbol (%) is used. Additionally, Mathematica lets you retrieve previous values using the cell input/output information by the % # command and the number of the cell or by explicitly writing the command with In [# of cell] or Out [# of cell]. As demonstrated in the next example, Mathematica gives the same value for each expression.

```
In[15]:=
%12
In[12]
Out[12]

Out[15]=  $\pi$ 
Out[16]=  $\pi$ 
Out[17]=  $\pi$ 
```

To determine whether a word is reserved within the Wolfram Language, use the Attributes command; this displays the attributes to the associated command. Attributes are general aspects that define functions in the Wolfram Language. When the word “Protected” appears in the attributes, it means that the word of the function is reserved. The next example shows whether the word “Power” is reserved.

```
In[18]:= Attributes[Power]
Out[18]= {Listable, NumericFunction, OneIdentity, Protected}
```

As seen in the attributes, “Power” is a protected word. Importantly, most of the built-in functions in Mathematica are listable—that is, the function is interleaved to the lists that appear as arguments to the function.

Variables can be presented in a notebook in the following ways: (1) global variables, or those that are defined and can be used throughout the notebook, like the ones in the earlier examples; and (2) local variables, which are defined in only one block that corresponds to what is known as a module, in which they are only defined within a module. A module has the following form: `Module [symbol1, symbol 2... body of module]`.

```
In[19]:= Module[{l=1,k=2,h=3},h Sqrt[k+1] + k + 1]
Out[19]= 3 + 3√3
```

Variables inside a module turn green by default; this is a handy feature for seeing the code inside a module block. A local variable only exists inside the module, so if you try to access them outside their module, the symbol is unassigned, as shown in the following example.

```
In[20]:= {l,k,h}
Out[20]= {l,k,h}
```

Variables can be cleared with multiple commands, but the most suitable command is the `Clear[symbol]`, which removes assigned values from the specified variable or variables. So, if you evaluate the variable after `Clear`, Mathematica treats it as a symbol, and you can check it with the command `Head`; `Head` always gives you the head of the expression, which is the type of object in the Wolfram Language.

```
In[21]:= Clear[a,x]
```

And if you check the head `a`, you see that “a” is a symbol.

```
In[22]:= Head[a]
Out[22]= Symbol
```

Symbols or variables assigned during a session remain in the memory unless they are removed or the kernel session ends.

**Note** Remove is an alternative to Clear.

---

## Built-in Functions

Built-in commands or functions are written in common English with the first letter capitalized. Some functions have abbreviations, while others employ PascalCase notation with two capital letters. Here, different examples of functions are presented. Built-in functions and group expressions often require arguments, which are values that the function needs to execute the correct operation. Functions may or may not accept arguments; they are separated by commas.

```
In[23]:= RandomInteger[]  
Out[23]= 0
```

---

**Note** RandomInteger, with no arguments, returns a random integer from the interval of 0 to 1, so do not panic if the result is not the same.

---

```
In[24]:= Sin[90 Degree] + Cos[0 Degree]  
Out[24]= 2
```

```
In[25]:= Power[2,2]  
Out[25]= 4
```

Built-in functions can also be assigned symbols.

```
In[26]:= d=Power[2,2]  
F=Sin[ $\pi$ ] + Cos[ $\pi$ ]  
Out[26]= 4  
Out[27]= -1
```

```
In[28]:= Clear[d,F]
```

Some commands or built-in functions in Mathematica have options that can be specified in a particular expression. To see whether a built-in function has available

options, use `Option`. In the next example, the `RandomReal` function creates a pseudo-random real number between an established interval.

```
In[29]:= Options[RandomReal]
Out[29]= {WorkingPrecision -> MachinePrecision}
```

`RandomReal` has only one option for specifying specific instructions within the `WorkingPrecision` command. The default value for this option is `MachinePrecision`. `WorkingPrecision` defines the number of digits of precision for internal computations, while `MachinePrecision` is the symbol used to approximate real numbers, denoted by `$MachinePrecision`. To see the value of `MachinePrecision`, type `$MachinePrecision`. The next example observes the difference between using default values for an option and employing custom values.

```
In[30]:= RandomReal[{0,1},WorkingPrecision->MachinePrecision]
RandomReal[{0,1},WorkingPrecision->30]
Out[30]= 0.19858
Out[31]= 0.451259323577871140781571594337
```

---

**Tip** In the Wolfram Language, global constants, which can be considered environmental variables, always start with a dollar sign (e.g., `$MachinePrecision`).

---

The first one returns a value with six digits after the decimal point, and the other returns a value with 30 digits after the decimal point. However, some built-in functions, such as `Power`, do not have any options associated with them.

```
In[32]:= Options[Power]
Out[32]= {}
```

## Dates and Time

The `DateObject` command provides results for concretely manipulating dates and times (see Figure 1-18). Date and time input and basic words are supported.

```
In[33]:= DateObject[]
Out[33]=
```



**Figure 1-18.** *The date of Wed 13 Sept 2023 and time zone*

DateObjects with no arguments give the current date, as shown in Figure 1-19. Natural language is supported in Mathematica—for instance, getting the date after Wed 13 Sept 2023.

```
In[34]:= Tomorrow
Out[34]=
```



**Figure 1-19.** *The date of Thu 14 Sep 2023*

The date format is entered as year, month, and day. It also supports string date formats and different calendars, as the next code dates show.

```
In[35]:= DateString[DateObject[{2020,6,10}]]
Out[35]= Wed 10 Jun 2020

In[36]:= DateString[DateObject[Today,CalendarType->"Julian"]]
DateString[DateObject[Today,CalendarType->"Jewish"]]
Out[36]= Wed 31 Aug 2023
Out[37]= Yom Revi'i 27 Elul 5783
```

The command also supports options that are related to a time zone.

```
In[38]:= DateString[DateObject[{2010,3,4},TimeZone->"America/Belize"]]
Out[38]= Thu 4 Mar 2010
```

Your current location's sunrise and sunset times can be calculated (support data is downloaded).

```
In[39]:= DateString[Sunset[Here,Now]]
DateString[Sunrise[Here,Yesterday]]

Out[39]= Wed 13 Sep 2023 18:41:27
Out[40]= Tue 12 Sep 2023 06:23:34
```



To get the current time, use `TimeObject` with zero arguments (see Figure 1-20). It can be entered in the format of 24h or 12h. To introduce the time, enter the hour, minute, and second.

```
In[41]:= TimeObject[]
Out[41]=
```



13:22:37

**Figure 1-20.** *Wed 13 Sep GMT-6 time*

Time zone conversion is supported; convert 5 p.m. from GMT-5 Cancun time to Pacific Time Los Angeles. You can also use `DateString` to use pure string objects.

```
In[42]:=
DateString[TimeZoneConvert[TimeObject[{17,0,0}],TimeZone-> "America/
Cancun"],"America/Los_Angeles"]]
Out[42]= 15:00:00
```

## Strings

Text can be useful when a description of the code is needed. Mathematica allows you to input text into cells and create a text cell related to your computations. Mathematica has different forms to work with text cells. Text cells can have lines of text, and depending on the purpose of the text, you can work with different text formats, like creating chapters, sections, or just general text. In contrast, to text cells, you can introduce comments to expressions that need an explanation of their purpose or just a description. For that, you simply write the comment within the symbols `(*)`. And the comments are shown with different colors; comments also always remain as unevaluated expressions. Comments can be single-line or multiline.

Mathematica can work with strings. To input a string, enclose the text in quotation marks “text”; Mathematica knows that it is dealing with text. Characters can be whatever you type or enter into the cells.

```
In[43]:= "Hello World" (*This is a comment*)
Out[43]= Hello World
```

Mathematica assumes that what you enter is text by being enclosed in quotation marks, although you can always impel it to explicitly treat it as text using the ToString command. You can check the head of the expression to make sure you are dealing with strings.

```
In[44]:= ToString[23.423563]
```

```
Out[44]= 23.4236
```

```
In[45]:= % // Head(*We use Head to know what type of object is*)
```

```
Out[45]= String
```

Strings appear without apostrophes when entered because it is the default format.

```
In[46]:= "Welcome to Mathematica"
```

```
Out[46]= Welcome to Mathematica
```

Whenever you put the type cursor over a string in Mathematica and enter input, it automatically appears surrounded by apostrophes. In this way, you can know you are working with strings.

Later, you learn about the functionality of AtomQ. The following demonstrates that strings cannot have subexpressions in the Wolfram Language. The output, true, indicates that the string input is a single, indivisible unit.

```
In[47]:= AtomQ["The sky is blue and tomorrow is expected to rain"]
```

```
Out[47]= True
```

You can also separate a string by characters.

```
In[48]:= Characters["Hello World"] (*Function that breaks the string into  
its characters*)
```

```
Out[48]= {H,e,l,l,o, ,w,o,r,l,d}
```

Replace particular characters in a string with a rule operator (→ or ->, in plain text).

```
In[49]:= StringReplace["Hello this is a string ",{"h","H"}->"4"] (*This  
function replaces the string each time it appears for rule of the  
pattern,that is 4*)
```

```
Out[49]= 4ello t4is is a string
```

Convert a text string to uppercase or lowercase.

```
In[50]:= ToUpperCase["hello my name is"]
```

```
Out[50]= HELLO MY NAME IS
```

```
In[51]:= ToLowerCase["HELLO MY NAME IS"]
```

```
Out[51]= hello my name is
```

Join a text string.

```
In[52]:= StringJoin["Nice", "to", "have", "you", "back"]
```

```
Out[52]= Nicetohaveyouback
```

Or with the string join symbol (<>).

```
In[53]:= "Nice"<>"to"<>"have"<>"you"<>"back"
```

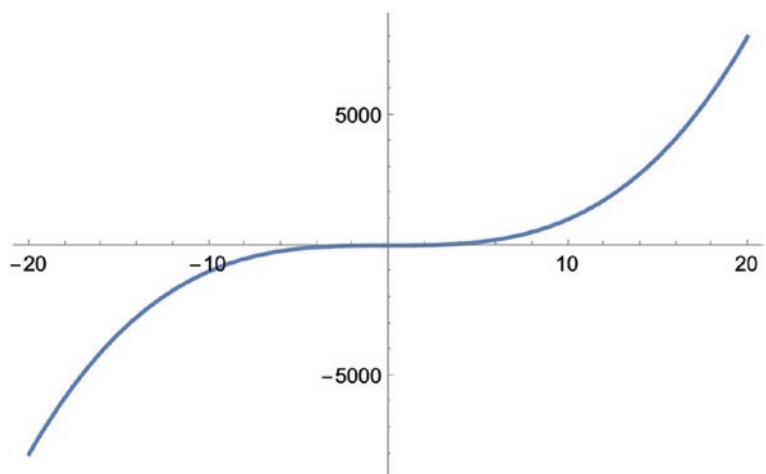
```
Out[53]= Nicetohaveyouback
```

## Basic Plotting

The Wolfram Language offers a basic description to easily create two-dimensional and three-dimensional graphics. It has a wide variety of graphics, such as histograms, contour, density, and time series. To graph a simple mathematical function, use the `Plot` command, accompanied by the variable symbol and the interval where you want to graph (see Figure 1-21).

```
In[54]:= Plot[x^3, {x, -20, 20}]
```

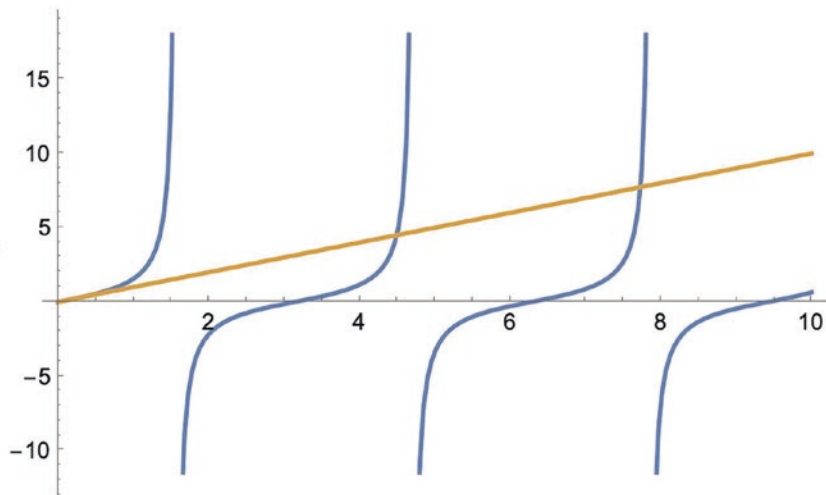
```
Out[54]=
```



**Figure 1-21.** *A cubic plot*

The plot function also supports handling more than one function; simply gather the functions inside curly braces. Figure 1-22 shows the two functions in the same graph; each with a unique color.

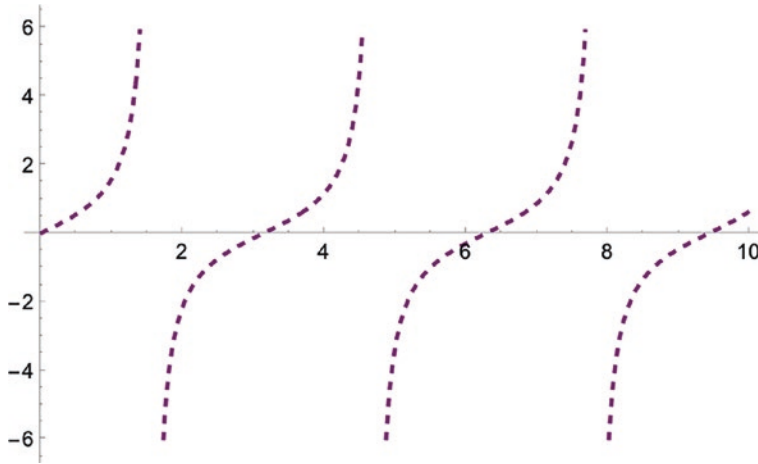
```
In[55]:= Plot[{Tan[x],x},{x,0,10}]
Out[55]=
```



**Figure 1-22.** *Multiple functions plotted*

You can also customize graphics in color if the curve is thick or dashed; this is done with the `PlotStyle` option (see Figure 1-23).

```
In[56]:= Plot[Tan[x],{x,0,10},PlotStyle->{Dashed,Purple}]
Out[56]=
```



**Figure 1-23.** *Dashed tangent function*

The `PlotLabel` option allows you to add basic descriptions to your graphics by adding a title. On the other hand, the `AxesLabel` option lets you add names to axes, both  $x$  and  $y$ , as depicted in Figure 1-24.

```
In[57]:= Plot[E^x,{x,0,10},PlotStyle->{Blue}, PlotLabel -> "e^x",AxesLabel->
{"x-axis","y-axis"}]
Out[57]=
```

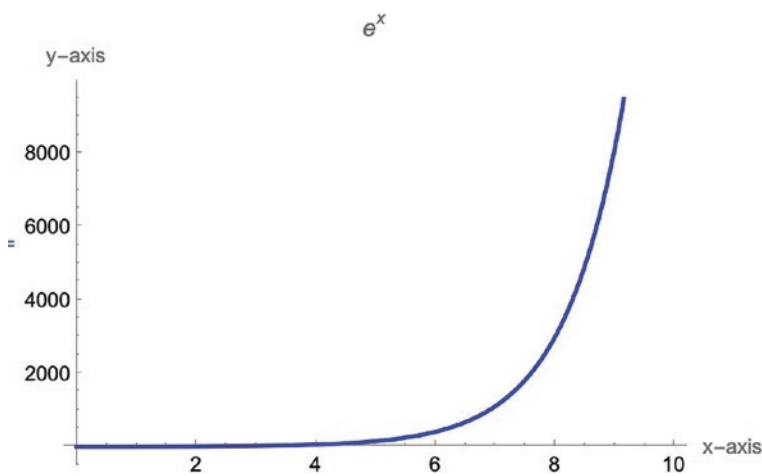


Figure 1-24. A plot with title and labeled axes

## Logical Operators and Infix Notation

Infix notation and logical operators are commonly used in logical statements or comparisons of expressions, and the result values are either true or false. Table 1-1 shows the relation operators of the Wolfram Language.

Table 1-1. Operators and Their Definitions

Definition	Operator Form
Greater than	>
Less than	<
Greater than or equal	≥
Less than or equal	≤
Equal	=
Unequal	!= or ≠
Structural Equality	===

Relational operators, also called *comparison operators* and *logical binary operators*, check the veracity or falsity of certain relationship proposals. The expressions that contain them are called relational expressions. They accept various types of arguments,

and the result can be true or false—that is, they are Boolean results. As you can see, they are all binary operators, of which two are of equality condition `==` and `!=`. These serve to verify the equality or inequality of expressions.

```
In[58]:= 6*1>2
```

```
Out[58]= True
```

```
In[59]:= 6*1<2
```

```
Out[59]= False
```

```
In[60]:= 1/2 >= 1/2
```

```
Out[60]= True
```

```
In[61]:= 1/4 <= 1/2
```

```
Out[61]= True
```

```
In[62]:= 3.12 == 2.72
```

```
Out[62]= False
```

```
In[63]:= π != √-1
```

```
Out[63]= True
```

```
In[64]:= 2===2.
```

```
Out[64]= False
```

Boolean operands produce a true or false result or test whether a condition is satisfied. Table 1-2 shows Boolean operators of the Wolfram Language.

**Table 1-2.** *Boolean Operators and Their Definitions*

Definition	Operator Form
AND	&& or $\wedge$
OR	or $\vee$
XOR	$\veebar$
Equivalent	$\Leftrightarrow$
Negation	$\neg$

The AND operator returns a true value if both expressions are true. Otherwise, the result is false.

```
In[65]:= 2==1 && 3.12==2
Out[65]= False
```

The OR operator returns true if any of the expressions is true. Otherwise, it returns false. This operator has an analogous operation to the previous one.

```
In[66]:= 2*2==3 || 23*2==1
Out[66]= False
```

The XOR operator is an exclusive “or” operator that returns true when both expressions differ. Otherwise, it returns false when the expressions have the same value.

```
In[67]:= 2==1 \[Xor] 2==2
Out[67]= True
```

The equivalent operator returns true if expressions are powered from each other. Otherwise, it returns false.

```
In[68]:= Power[1,2] \[Equivalent] 1^2
Out[68]= True
```

The negation operator, also called logical negation, returns a value that can be an expression that evaluates to a result. The result of this operator is always a Boolean type.

```
In[69]:= \[Not]2==1
Out[69]= True
```

Another approach, instead of using Boolean operators, is to use different functions with postfix (Q), which consists of testing whether an object meets the condition of the built-in function. A few honorable mentions are SameQ, UnsameQ, AtomQ, IntegerQ, and NumberQ. The next example tests whether a number is a float expression or an integer.

```
In[70]:=
IntegerQ[1]
IntegerQ[1.]
Out[70]= True
Out[71]= False
```



The valuable application of the `AtomQ` function can tell you whether an expression is subdivided into subexpressions. Later, you are shown how to deal with subexpressions with lists. If the result is true, then the expression cannot be subdivided into subterms, and if it is false, then the expression has subterms.

```
In[72]:= AtomQ[12]
Out[72]= True
```

As shown, numbers cannot be subdivided because a number is a canonic expression; the same applies to strings, as seen before.

## Algebraic Expressions

The Wolfram Language can work with algebraic expressions. For instance, perform symbolic computations, algebraic expansions, and simplifications. Many words used in common language in algebra are preserved in Mathematica. To expand an algebraic expression, use `Expand`.

```
In[73]:= Expand[(x^2+y^2)*(x+y)]
Out[73]= x^3+x^2 y+x y^2+y^3
```

Adding a space between variables is the same as adding the multiplication operator. This can be checked by `a*x==a x`.

```
In[74]:= Expand[a x^2*(a x)^3]
Out[74]= a^4 x^5
```

But be careful when writing algebraic expressions because the `ax` symbol is not the same as an `x`. This also is checked using the `SameQ[ax, a x]` or the short notation `a x === ax`.

To simplify an expanded expression, use `Simplify` or `FullSimplify`.

```
In[75]:= Simplify[x^3+x^2 y+x y^2+y^3]
FullSimplify[x^3+x^2 y+x y^2+y^3]
Out[75]= x^3+x^2 y+x y^2+y^3
Out[76]= (x+y) (x^2+y^2)
```

The difference is that the latter tries transformations to simplify the expression more broadly. To unite terms over a repeated denominator, use `Together`. To expand into partial fraction decomposition, use `Apart`.

```
In[77]:=
Together[ $\frac{1}{z} + \frac{1}{z+1} - \frac{1}{z+2}$ ]

Apart[ $\frac{2+4z+z^2}{z(1+z)(2+z)}$ ]

Out[77]= (2+4 z+z^2)/(z (1+z) (2+z))
Out[78]= 1/z+1/(1+z)-1/(2+z)
```

## Solving Algebraic Equations

Various functions are accessible for finding solutions to algebraic equations. The most common is the `Solve` function. The first argument is the equation or expression to be solved, and the second is for the variable to be solved.

```
In[79]:= Solve[z^2+1==2,z]
Out[79]= {{z → -1},{z → 1}}
```

---

**Note** As you might remember, equal is expressed as double equal (`==`); do not use one equal (`=`) because that means assigning a value to a symbol or variable.

---

The result means that  $z$  has two solutions: one is  $-1$ , and the other is  $1$ . Each result is expressed in the form of a rule. A rule expression changes the assignment of the left side to the one on the right side (left  $\rightarrow$  right) whenever it applies. For example,  $z \rightarrow 1$  is the same as `Rule[z, 1]`.

To verify the solution, the values of  $z$  ( $-1, 1$ ) must be replaced in the original equation. For this, you can use the `ReplaceAll` operator (`/.`) along with the rule command  $\rightarrow$  or `Rule`, which is used to apply a transformation to a variable or a pattern with other expressions.

```
In[80]:= z^2+1 /.Rule[z,{1,-1}]
Out[80]= {2,2}
```

The other option is to type the solutions explicitly in the equation.

```
In[81]:= {1^2+1==2, (-1)^2+1==2}
```

```
Out[81]= {True,True}
```

Multiple equations can be solved, too, given a system of equations and a list of interested variables. To solve the equations, place the system of equations in one list and the variables in another.

For example, solve the next system of equations.

$$x + y + z == 2$$

$$6x - 4y + 5z == 3$$

$$5x + 2y + 2z == 1$$

The solution is

```
In[82]:= Solve[{x+y+z == 2, 6x-4y+5z == 3, x+2y+2z == 1},{x,y,z}]
```

```
Out[82]= {{x -> 3, y -> 10/9, z -> -19/9}}
```

---

**Note** The results are listed. Lists are essential structures in the Wolfram Language and are discussed in the next chapter.

---

The latter process is also applicable to equations assigned to variables. You can write this with the use of compound expressions.

```
In[83]:=
```

```
EQ1=x+y+z==2;
```

```
EQ2=6 x-4 y+5 z==3;
```

```
EQ3=x+2 y+2 z==1;
```

```
Solve[{EQ1,EQ2,EQ3},{x,y,z}]
```

```
Out[83]= {{x -> 3, y -> 10/9, z -> -19/9}}
```

The Solve function also works with pure algebraic equations.

```
In[84]:= Solve[{x + y + z == a, 6 x - 4 y + 5 z == b, x + 2 y + 2 z == c},
{x, y, z}]
```

```
Out[84]= {{x -> 2a - c, y -> 1/9(7a - b - c), z -> 1/9(-16a + b + 10c)}}
```

The Solve function supports expressions with a mixture of logical operators, expressing y and x in terms of z.

```
In[85]:= Solve[EQ1 && EQ2, {x, y}]
```

```
Out[85]=  $\left\{ \left\{ x \rightarrow \frac{1}{10}(11-9z), y \rightarrow \frac{9-z}{10} \right\} \right\}$ 
```

It also uses the OR operator.

```
In[86]:= Solve[x^2 + y^2 == 0 || x - 2 y == 1, x]
```

```
Out[86]=  $\{\{x \rightarrow -iy\}, \{x \rightarrow iy\}, \{x \rightarrow 1 + 2y\}\}$ 
```

The Solve function returns the solution for each of the equations entered.

Establishing a condition with the AND operator lets you return solutions that satisfy a condition; for example, the following equation has two solutions 1 and -1, but you can solve the equation with the condition that z must be different from 1.

```
In[87]:= Solve[z^-2 + 1 == 2 && z != 1, z]
```

```
Out[87]=  $\{\{z \rightarrow -1\}\}$ 
```

To obtain more general results, Reduce is used, as shown in the following example.

```
In[88]:= Reduce[Cos[x]==-1,x]
```

```
Out[88]=  $c_1 \in \mathbb{Z} \ \& \ (x = -\pi + 2\pi c_1 \ || \ x = \pi + 2\pi c_1)$ 
```

Here, the alternative solutions are separated by the OR operator, and the condition is established by the AND. So this means that there are two possible solutions  $-\pi + 2\pi c_1$  or  $\pi + 2\pi c_1$  and that the constant  $c_1$  must be a number that belongs to the integers ( $\mathbb{Z}$ ). In addition, Reduce can also solve inequalities.

```
In[89]:= Reduce[h^2+k^2<11,{h,k}]
```

```
Out[89]=  $-\sqrt{11} < h < \sqrt{11} \ \& \ -\sqrt{11-h^2} < k < \sqrt{11-h^2}$ 
```

Here, the simultaneous equations are for h and k. Furthermore, Reduce can show the combination of equations with certain conditions.

```
In[90]:= Reduce[ $\alpha + \beta * \alpha^2 = E, \alpha$ ]
```

```
Out[90]=  $(\beta == 0 \ \& \ \alpha == e) \ || \ \left( \beta \neq 0 \ \& \ \left( \alpha = \frac{-1 - \sqrt{1 + 4e\beta}}{2\beta} \ || \ \alpha = \frac{-1 + \sqrt{1 + 4e\beta}}{2\beta} \right) \right)$ 
```

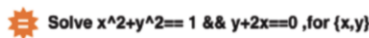
The first solution is that  $\alpha$  and  $\beta$  must be the number  $e$  and zero. The second solution is in terms of  $\alpha$  and the condition that  $\beta$  must differ from zero.

## Using Wolfram Alpha Inside Mathematica

A really good application inside Mathematica uses the Wolfram Alpha computable knowledge base. Wolfram Alpha can be called from Mathematica with the Wolfram Alpha query. To enter the Wolfram Alpha query, type the double equal sign before typing any expression; an orange asterisk with a white equal sign should appear, meaning that the input typed is a query with natural language. To execute the cell, click the Enter key.

So, for example, algebraic equations can be solved using the Wolfram Alpha query. Type the double equal sign (`==`) in an input cell, and the Wolfram Alpha query symbol should appear (see Figure 1-25). Alternatively, select Wolfram Alpha query as a new input from the `+` menu (left of the horizontal line) for a new cell.

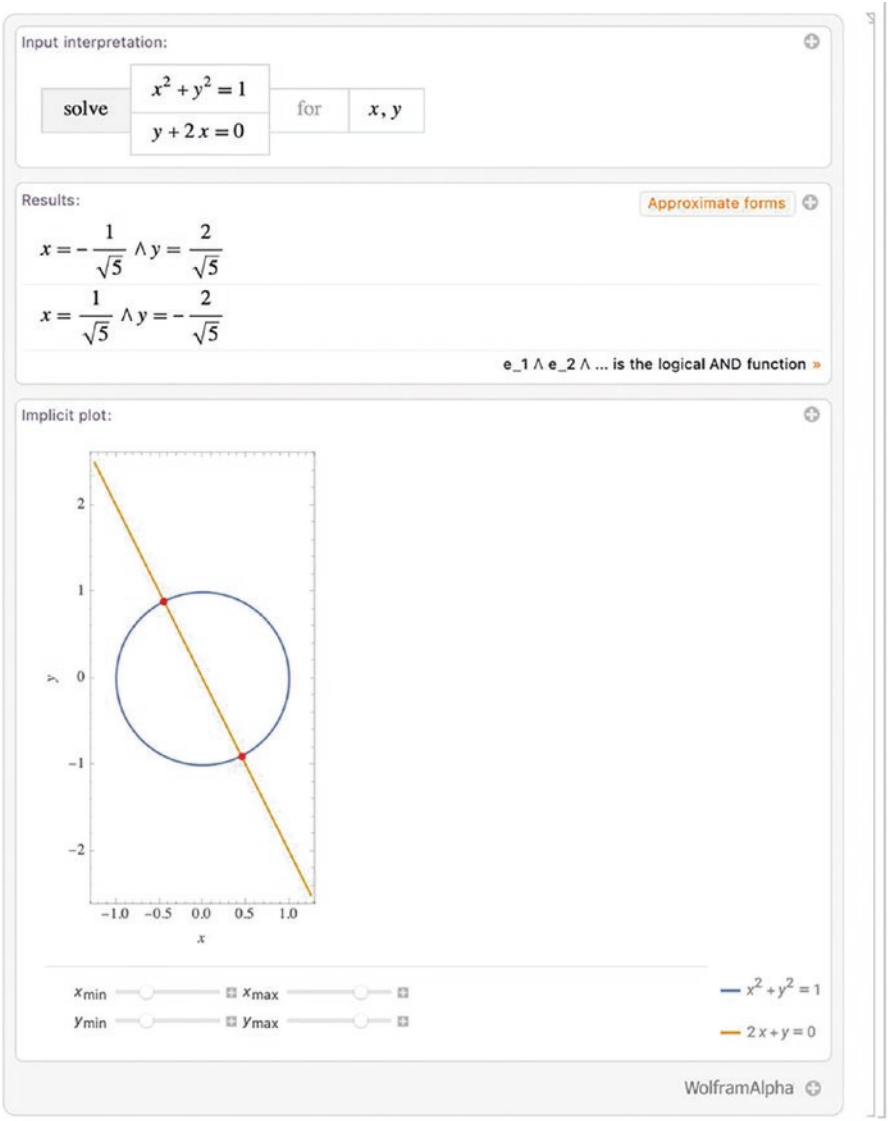
In[91]:=



**Figure 1-25.** Wolfram Alpha query input

Out[91]=

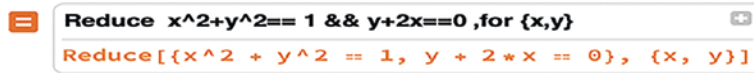
Figures 1-25 and 1-26 show the input and output of the Wolfram Alpha query.



**Figure 1-26.** Wolfram Alpha query output

As shown, the system returns the solutions for x and other calculations. The cell represents the calculations in the Wolfram Alpha form. Clicking the plus icon shows a list of different forms of input. To see the equivalent in the Wolfram Language, select the Input option. The other related way to use Wolfram Alpha is with free-form input. It is worth mentioning that words associated with Mathematica commands, like Reduce, can be used too. Figure 1-27 shows the input cell in the free-form input. Clicking the plus

icon shows more calculations, like in the Wolfram Alpha query. The following code is the equivalent in the Wolfram Language of input typed. Clicking the code, replace it with the Wolfram Language syntax.



**Figure 1-27.** Input code in the free-form input

In[91]:=

$$\text{Out}[91]= \left( x == -\frac{1}{\sqrt{5}} \parallel x == \frac{1}{\sqrt{5}} \right) \& \& y == -2x$$

A clarification here, not just calculations can be made. With Wolfram Alpha, access to curated data for various topics is available; for example, getting the financial data for a particular stock in March (see Figure 1-28) or the population of Australia, as depicted in Figure 1-29.

In[93]:=

Out[93]=



**Figure 1-28.** Input and output of the Tesla stock in March 2023. Identified by the financial entity and returns a TimeSeries object, made possible by the latest version of Mathematica

In[94]:=

Out[94]=



**Figure 1-29.** Input for the population of Australia

Both free-form input and Wolfram Alpha queries can be useful and practical tools. For example, if you do not know the appropriate syntax of a function or command, try using the free-form input in natural language so that, when evaluated, you can get the equivalent Wolfram Language syntax of that function. Nevertheless, a downside of the Wolfram Alpha query is that the computations are done outside Mathematica, meaning that the computations are made on the Wolfram Alpha servers. In contrast, calculations with free-form input can be reproduced inside Mathematica. Sometimes it is preferable to work directly with the Wolfram Language to better manage the results, as extracting results from Wolfram Alpha can be tedious. It should be noted that to access these two features from Mathematica, it is necessary to have access to Wolfram servers via an online network.

## Delayed and Immediate Expressions

The Wolfram Language has two important features. First, let's look at how the Set mechanism works. The symbol `=` is the script for Set, and `:=` is for SetDelayed. The Set mechanism is represented by `W = expr`. `W` is the variable you are assigning a value to, and `expr` is the expression or value you are assigning to `W`. This means that Mathematica evaluates the expression straightaway, then each time the variable or defined function is called, the value of `W` is written, and the result is shown. On the contrary, using `W := expr` means that the expression is not evaluated until called, so each time the `W` is called, it evaluates the stored expression every time.

```
In[95]:= W=RandomReal[]
Out[95]= 0.536369
```

Test whether `W` equals `W`.

```
In[96]:= W==W
Out[96]= True
```

The condition is true in this case because Set is used for declaring the `W` variable with the `RandomReal` function, which returns a pseudo-random choice from 0 to 1. The same approach is used for SetDelayed, and the result is false because every time `W` appears, the function is called for a new evaluation. You can write the code as a compound expression.



```
In[97]:= Clear[W];W:=RandomReal[];W
Out[97]= 0.550058
```

Let's check.

```
In[98]:= W==W
Out[98]= False
```

The result is false since the `RandomReal` function is evaluated again each time `W` is called. So, the first `W` evaluates `RandomReal`, and the second `W` again evaluates `RandomReal`, even though they are the same symbol.

The same approach applies to `Rule` ( $\rightarrow$ ) for immediate evaluation and `RuleDelayed` ( $\rightarrow$ ) for evaluation only when used. Consider the following example.

```
In[99]:=
z=2; (*Assigning 2 to z*)
R=z->z^3; (*Rule example*)
RD=z:>z^3; (*RuleDelayed example*)
R
RD
Out[102]= 2 → 8
Out[103]= 2 :> z^3
```

The expression returns  $2 \rightarrow 8$  since `z` is evaluated immediately, while the expression `z :> z^3` delays the evaluation of `z^3` until it is applied. These operators can be used with the `ReplaceAll` operator (`/.`) as previously seen with algebraic equations.

## Improving Code

Code efficiency is essential to achieve performance and decrease resource consumption, leading to faster execution times and improved maintenance. One specific context where these matters are improving code for increased efficiency and reliability in Mathematica and Wolfram Language. As a developer, you can achieve greater readability and facilitate easier troubleshooting by using the built-in functions. Also, built-in symbols are optimized for efficiency, making them preferable to defining your own.

## Code Performance

In Mathematica, there are many ways to write an expression in the same form. However, when you carry out long code operations, there may be a better notation to improve the performance of the code and thus not consume too many computational resources. This can be achieved by the relative performance of different functions for the development of the same result. The Wolfram Language provides a measure of this. The timing function shows the performance in units of seconds to each process in relation to the value of `$TimeUnit`, which is the CPU time it takes for the Wolfram Language kernel to carry out the process. `$TimeUnit` varies from system to system, so you might get something different—such as 1/1000.

---

**Note** A lower value of `$TimeUnit` would be considered more precise than using it, as it provides a higher granularity or resolution in the time measurements.

---

The following example shows how long it takes to calculate the expression with a built-in function and a common power expression. Timing returns two values: the unit time and the calculation result, but the output is suppressed because it is a very big value.

```
In[104]:= Timing[Power[10,10^8];]
```

```
Out[104]= {1.1401,Null}
```

```
In[105]:= Timing[10^10^8;]
```

```
Out[105]= {1.54863,Null}
```

As you see, there is a difference between each; this has to do with how the Wolfram Language processes each computation and your computer specs. To look at the absolute

```
In[106]:= AbsoluteTiming[10^10^8;];]
```

```
Out[106]= {1.13833,Null}
```

```
In[107]:= AbsoluteTiming[Power[10,10^8];]
```

```
Out[107]= {1.13189,Null}
```

There is a difference, too, as in the case with Timing. To restrain a computation by time, use `TimeConstrained`. With this command, time constraints can be added to a

calculation. The evaluation is aborted if the code is still running and the time limit has been reached. For example, abort the evaluation after 1 second has passed.

```
In[108]:= TimeConstrained[10^10^8,1]
Out[108]= $Aborted
```

The EchoTiming function has been improved and can display the timing information of an evaluated expression. EchoTiming supports the latter methods of Timing and AbsoluteTiming.

```
In[109]:=
EchoTiming[Power[10,10^8];,"Time in seconds:",Method->Timing]
EchoTiming[Power[10,10^8];,"Time in seconds:",Method->AbsoluteTiming]
Out[109]= Time in seconds: 1.13813
Out[110]= Time in seconds: 1.12619
```

## Handling Errors

Mistakes may be commonplace, as you most commonly develop code as you continue to learn. When a function fails, Mathematica displays a message below the written function. The message form provides the name of the function associated with the error, along with a possible description of the cause of the error.

Next, let's look at how this works (see Figure 1-30).

```
In[111]:= StringJoin["hello","I am ",Jeff]
Out[111]= helloI am <>Jeff
```

 **StringJoin**: String expected at position 2 in helloI am <> Jeff.

**Figure 1-30.** Error message for the code entered

The associated function in the message appears in red (see Figure 1-20). What happens here is that the StringJoin function works only for strings, and you are writing a Jeff variable, not a string, hence the error.

To learn more about the error, click the red ellipsis icon. A menu appears, listing the different options available to handle the error. To review the error in the documentation, you must click the error option, which is the option that has an open book icon. This option takes you to the documentation of the associated function.

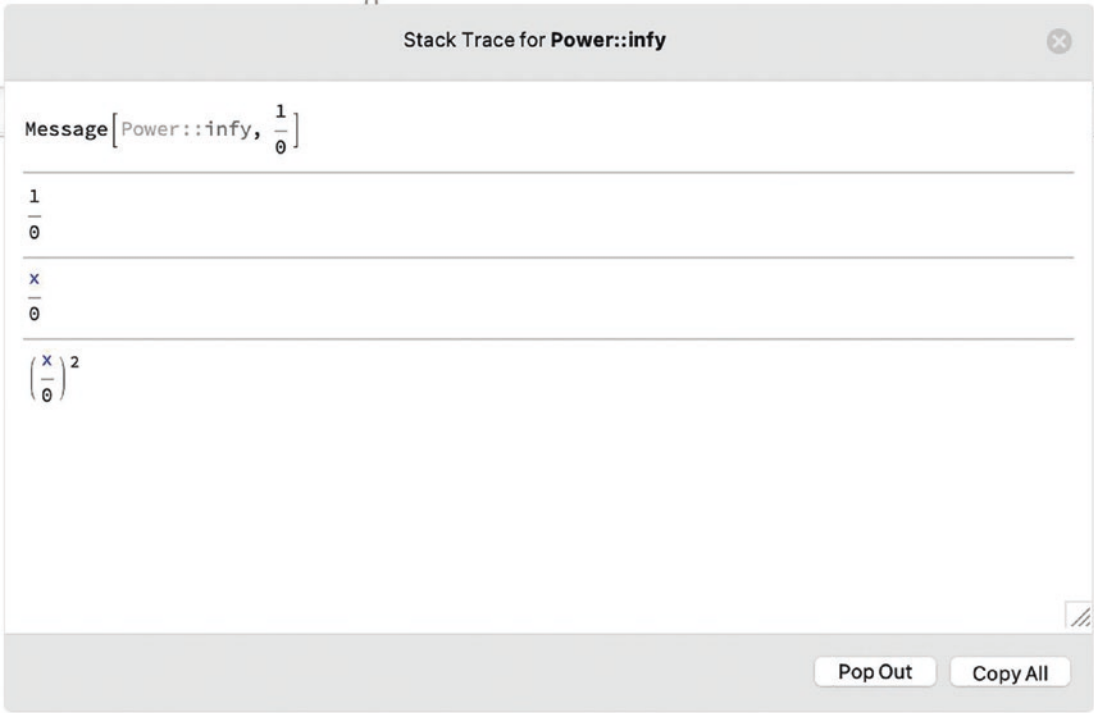
Another option from the pop-up menu that appears is Show Stack Trace. This option shows you graphically and in blocks how the function and its expressions are being evaluated. This option is analogous to the Trace command. Let's look at the next example error and Figure 1-31.

```
In[112]:= Power[x/0,2]
Out[112]= ComplexInfinity
```



**Figure 1-31.** Error message for infinite expression

Here, the error is that Mathematica encounters a division by zero, which is undefined, and you can see the trace of the function with Stack Trace in Figure 1-32.



**Figure 1-32.** Show Trace Stack pop-up window

## Debugging Techniques

In Mathematica, debugging practices help programmers identify, diagnose, and fix errors or unusual behavior in their written segments of code. Traditional code operations using the Wolfram Language built-in functions like `Trace`, `Echo`, and `Print`, among others, let you follow each step of your code as it runs. This makes it easier to focus on the specific implementation details and not the whole abstract operations that the code does, providing a flexible and robust sense of what the code or code block should do.

Since version 13, a few improved built-in functions, like `EchoLabel` and `EchoEvaluation`, have been added to the repertoire, as seen in the following example.

```
In[113]:=
x=2;
Echo[x];x=x^2+1;
Echo[x];x=x^2+1;
Echo[x];

Out[114]=
>> 2
>> 5
>> 26
```

Let's go over what happened here. Initially, the value 2 is assigned to `x`. The first `Echo` prints the value of `x`, which is 2. Then, in the 2nd operation, `x` is updated based on its original form. Subsequently, the second `Echo` prints the new value, 5, which continues until the final value of 26 is reached ( $5^2 + 1$ ).

The same can be achieved using `EchoLabel` and `EchoEvaluation` but tagging costume messages.

```
In[117]:=
x=2;
Echo[x,"Initial Value: "];
x=x^2+1;
EchoLabel["First Iteration: "][x];
x=x^2+1;
EchoEvaluation[x=x^2+1,"Second Iteration: "->"Output :"];
```

```

Out[118]=
>> Initial Value:    2
>> First Iteration:   5
<< Second Iteration:  x=x^2+1
<< Output :    677

```

The previous example performs three iterations of the same operation on the same initial value. The first Echo prints the value of  $x$ . The second EchoLabel prints the output of the first iteration with a costume label and finalizes with the last evaluation and label association. Before evaluation, the initial label is printed, followed by the second label being printed once the evaluation is complete. Throughout the process, it displays results next to symbols with different colors: orange ( $>>$ ) and blue ( $<<$ ). The first symbol represents output, and the second symbol represents input.

Now, by utilizing operations to measure the time, as seen before, you can combine them to pinpoint which stages demand more time to compute, as exemplified in the following example.

```

In[123]:=
x=2;
EchoTiming[Echo[x,"Initial Value: "]];
x=x^2+1;
EchoTiming[EchoLabel["First Iteration: "][x]];
x=x^2+1;
EchoTiming[EchoEvaluation[x=x^2+1,"Second Iteration: "->"Output :"]];
Clear[x];

Out[124]=
>> Initial Value:    2
⌚ 0.013603
>> First Iteration:   5
⌚ 0.018695
<< Second Iteration:  x=x^2+1
>> Output :    677
⌚ 0.031909

```

As seen, the last evaluation took the longest time (0.031909 seconds), while the initial value estimation was the fastest (0.013603 seconds). These techniques are useful when program flow is broken into small chunks of digestible code, like visualizing variable values at key points and gauging computation time for performance breakdown.

## How Mathematica Works

This section explores the internal workings of computations and discovers ways to visualize data using multiple basic yet powerful commands.

### How Computations are Made (Form of Input)

Each time Mathematica receives a computation in the input cell, it uses the `StandardForm`, which is the output representation of expressions in the Wolfram Language and has many aspects of common mathematical notation. Input can be written in various forms, but to know how the expression is written in the Wolfram Language, `StandardForm` is used.

```
In[130]:= StandardForm[1/x+x^2]
```

```
Out[130]//StandardForm=
```

$$\frac{1}{x} + x^2$$

`InputForm` works similarly but produces the output acceptable to be entered as Wolfram Language input.

```
In[131]:= {InputForm[1/x + x^2], InputForm[a^x], InputForm[a_x], InputForm[Sqrt[2]]}
```

```
Out[131]= {x^(-1) + x^2, a^x, Subscript[a, x], Sqrt[2]}
```

Every type of format has its equivalent in one line of code text, like the square root symbol ( $\sqrt{\phantom{x}}$ ), which means the same as `Sqrt[ ]`. To convert input into `StandardForm`, `InputForm`, and other forms, select the cell block and head to **Cell** ► **Convert To** ► `StandardForm`, and `InputForm`, among others. `StandardForm` and `InputForm` apply to every expression in the Wolfram Language. Try using `InputForm` on the previous plots to see how the expression is written completely. To understand better how Mathematica works, you want to know how symbolic or numeric computations are performed or written. The `FullForm` and `TreeForm` commands can be applied to view how expressions

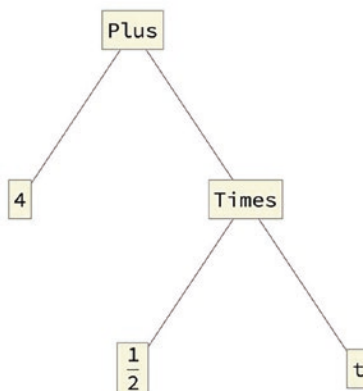
are represented symbolically. `TreeForm` represents the command in a graphical format, while `FullForm` represents the form of the expression managed internally by the Wolfram Language.

```
In[132]:= FullForm[t/2 + 2^2]
Out[132]//FullForm= Plus[4,Times[Rational[1,2],t]]
```

`FullForm` also represents the input as a one-line output code, like `InputForm`. But even if `InputForm` also returns a one-line output code, why not use `InputForm`? The reason is that `FullForm` represents what Mathematica understands as input. With this in mind, `FullForm` is useful because it lets you know what Mathematica interprets about the written input. In Mathematica, the mathematical order of operations is preserved. So the previous output is as follows: first, Mathematica detects the rational number  $1/2$  (`Rational[1,2]`) and the symbol `t`, followed by the multiplication of these two elements (`Times[Rational[1,2],t]`) followed by the addition of 22 (`Plus[4, Times[Rational[1,2],t]]`).

Another type of command that helps in creating a visualization of how Mathematica manipulates expressions is `TreeForm`. `TreeForm` returns the expression as a tree plot (see Figure 1-33). Alternatively, you can apply commands using the postfix form ‘`expr // function`’, rather than writing in the canonical form ‘`F[expression]`’.

```
In[133]:= t/2 + 2^2 // TreeForm
Out[133]//TreeForm=
```



**Figure 1-33.** Tree plot representation



In short terms, Mathematica detects the multiplication of  $1/2$  times  $t$  and then proceeds to add the result of the product with the result of two squared. The tree graph is read from bottom to top until you reach the top of the tree.

One more helpful command is `Trace`. `Trace` returns individual forms corresponding to the evaluation line, which contains the sequence of forms of the evaluated expression.

```
In[134]:= Trace[Plus[4,Times[Rational[1,2],t]]]
```

```
Out[134]= {{Rational[1,2], 1/2}, {t/2}, 4+t/2}
```

So here, the sequence of operations is as follows: first use the term `Rational[1, 2]`, followed by  $1/2$ , then  $1/2$  is multiplied by  $t$ , and the result is added to 4. Using `FullForm` in `Trace` lets you see how the internal structure changes.

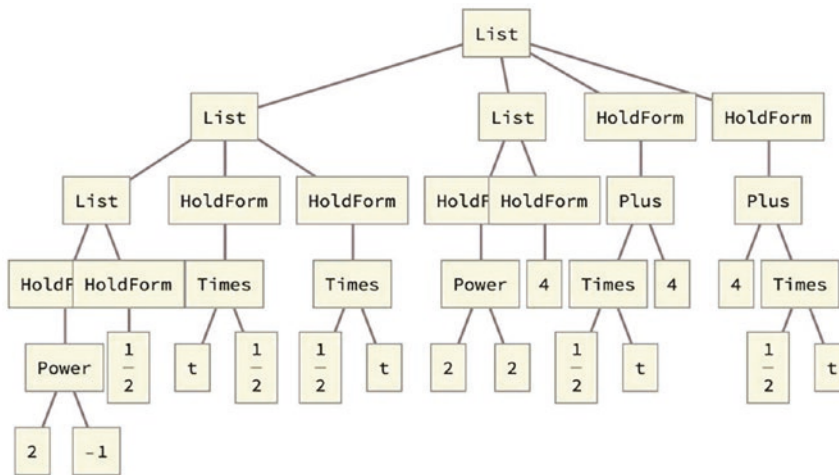
```
In[135]:= FullForm[Trace[Plus[4,Times[Rational[1,2],t]]]]
```

```
Out[135]//FullForm= List[List[List[HoldForm[Rational[1,2]],HoldForm[Rational[1,2]]],HoldForm[Times[Rational[1,2],t]]],HoldForm[Plus[4,Times[Rational[1,2],t]]]]
```

It can be seen that the terms change each step. The `HoldForm` command is used to see the output in an unevaluated form. As a complement to `Trace`, `FullForm` and `TreeForm` can be combined to see the hierarchy of operations in an expression internally, as seen in Figure 1-34.

```
In[136]:= Trace[t/2+2^2]//TreeForm
```

```
Out[136]//TreeForm=
```



**Figure 1-34.** *TreeForm and Trace combined*

Here, the tree shows how changes are made and read from left to right. Reading the tree, you see that Mathematica recognizes that  $1/2$  is  $2^{-1}$ ; this is followed by  $t$  times  $1/2$ , followed by  $2^2$ , which is 4, and so on until the end. Moving the cursor over each block displays a representation of the operation being held. There may be occasions when you encounter operations or expressions you do not understand. A solution to this would be using the previous commands, which allow you to see the expression's inner structure and thus understand how the operation is performed.

## Searching for Assistance

The Wolfram Documentation Center contains the registry of all built-in functions. Documentation of functions can be accessed through the front end by opening a new window, clicking the Help tab on the toolbar, or entering expressions. Since version 13.1, the documentation can also be accessed through the toolbar's rightmost icon, which is an open book icon. The Input Assistant is displayed as an autocomplete or suggestion bar when a command or related sensible options are written. When writing a built-in function or command, Mathematica tries to automatically complete the phrase.

Like in Figure 1-35, type the word **Random**, and different associated commands appear as suggestions. If the desired command is listed, you can select it with the cursor pointer.



**Figure 1-35.** Autocomplete pop-up menu

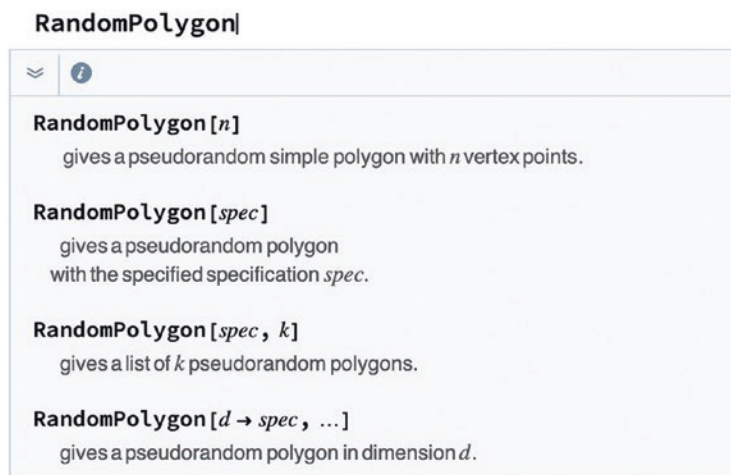
To access the documentation for a particular command, click the “i” document icon next to the command name, and the documentation windows should appear.

---

**Note** Autocomplete also works for assigned symbols.

---

When writing the built-in function or command followed by the left square bracket, the completion menu appears; if you click the double-down arrow, it displays the input forms supported by that command, as shown in Figure 1-36.

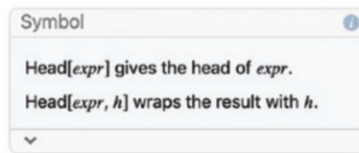


**Figure 1-36.** Built-in function *RandomPolygon* with different input forms

As seen in the example, the *RandomPolygon* function has four types of input forms; also, in the menu, you can see text related to the different forms of the input.

To learn how a function works or how built-in functions are written, the best resource is to consult the Wolfram Documentation Center. You can also check if an alternative input expression can be used. So, if you need help understanding how the `Head` function works, you input a question mark (?) before the function's name, giving you a simple understanding of how the command works (see Figure 1-37). If you want additional information related to the attributes of the function, a double question mark (??) can be employed. As a piece of advice, the Wolfram Documentation Center can be used for more in-depth options. Use the F1 shortcut, which opens the Documentation Center. If you highlight the symbol name and press F1, you are taken directly to the documentation page for that symbol.

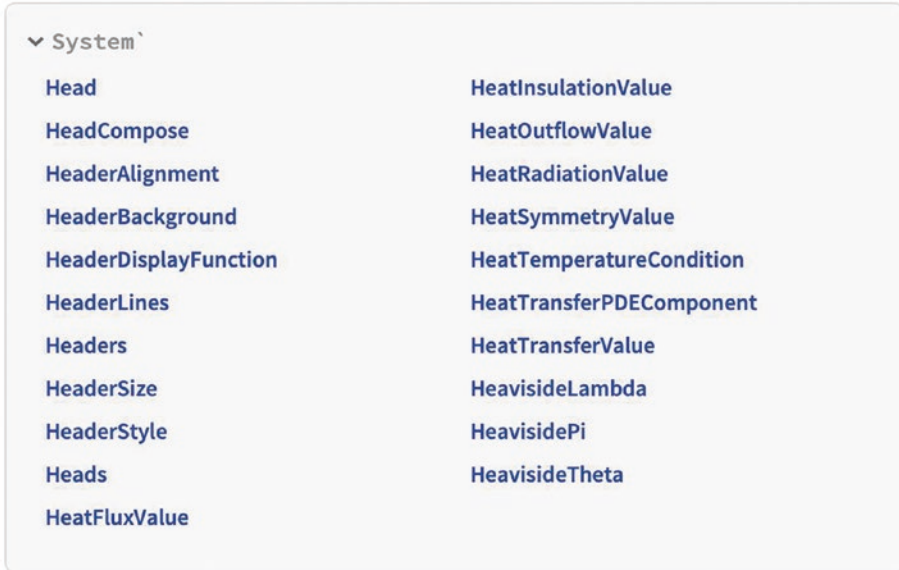
```
In[137]:= ?Head
Out[137]=
```



**Figure 1-37.** Output information for the `Head` command

The previous command showed how to show information related to a specific function. But if you don't recall the exact spelling, you can write the first letters of the name followed by an asterisk (\*), and Mathematica provides a list that matches your query. In the following example, the output is the functions whose names start with "Hea" (see Figure 1-38). The Wolfram documentation can be used in a scenario that needs more in-depth knowledge.

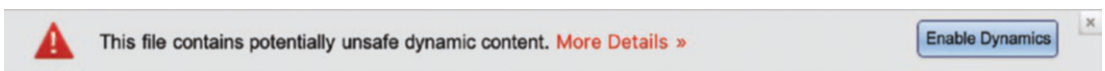
```
In[138]:= ?Hea*
Out[138]=
```

**? Hea\***

**Figure 1-38.** Output information for the commands starting with the letters Head

## Notebook Security

The Wolfram Language provides creation and the ability to run dynamic content. These contents allow the user to create programs that can perform useful and complex tasks; on certain occasions, unwanted content may be executed or code misused. A notebook may or may not contain dynamic content as part of its code. Notebooks containing dynamic content can be instantly downloaded without any user action. Sometimes, Mathematica alerts the user when a notebook contains dynamic content, displaying a message like that shown in Figure 1-39.



**Figure 1-39.** Warning message of dynamic content

If the notebook is not found in a trusted directory, a message warns the user that the notebook contains unreliable dynamic content. The dynamic content is executed without displaying a previous message to the user if the notebook is located in a

reliable directory. To find out if a notebook is located in a trusted directory with the name `TrustedPath`, check out the trusted math directories, which are found in (1) `$BaseDirectory`, (2) `$UserBaseDirectory`, and (3) `$InitialDirectory`.

```
In[139]:= $BaseDirectory
```

```
Out[139]= /Library/Mathematica
```

```
In[140]:= $UserBaseDirectory
```

```
Out[140]= /Users/macosex/Library/Mathematica
```

```
In[141]:= $InitialDirectory
```

```
Out[141]= /Users/macosex
```

In this case, these are the trusted directories; yours may defer. By default, the directories called `UntrustedPath` are those from which you can store files that can be potentially harmful, such as files downloaded from the Internet. For this, in the Wolfram Language, the user's writing directories and configuration directories are called `UntrustedPath`. To add, change, or remove the trusted and untrusted directories, go to the Preferences menu and then to the Security tab. There are options to edit unreliable and trusted directories.

## Summary

This chapter served as an introduction to Mathematica, a comprehensive software used for mathematical computation and analysis. The chapter also introduced the unique Wolfram Language used within the software, focusing on its notebook interface, text processing, palettes, and various styles and features. It also delved into expressions in Mathematica and concluded with topics related to code performance, error and debugging management, and ensuring security.

## CHAPTER 2

# Data Manipulation

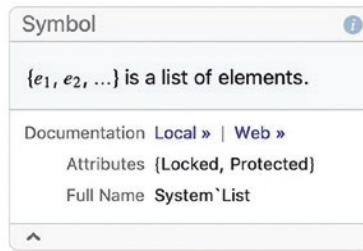
This chapter reviews the basics of data creation and data handling in the Wolfram Language. The chapter begins with the concept of lists and structures within the language. Numbers, digits, and simple ways to use them with common math functions are discussed. Next, you are introduced to lists of objects, representing, and generating lists, delving into data arrays and examining nested lists, vectors, matrixes, and relevant operations for various purposes. The chapter ends with study list manipulation techniques—retrieving, assigning, or removing data—and structuring lists to offer a general guide to understanding list manipulation in the Wolfram Language.

## Lists

Lists are the core of data construction in the Wolfram Language. Lists can gather objects, construct data structures, create tables, store values or variables, make elementary to complex computations, and characterize data. A list can represent any expression in the Wolfram Language (numbers, text, data, images, graphics, etc.)—that is, any set of whichever data.

If you access the information structure of a list, as demonstrated in Figure 2-1, you can see the typical format to form a list. Lists are represented by curly braces or the List command. In the Wolfram Language, almost every data object result can be listable; in other words, lists allow you to group data that maintain some type of relationship, even if they are of a different type, by manipulating all together (using the same identifier) or each separately.

```
In[1]:= ??List  
Out[1]=
```



**Figure 2-1.** List definition in the Wolfram Language

As seen in the evaluation, commas separate elements, and the whole list is between curly braces. Also, List is a protected variable, meaning you cannot assign values to the name List.

## Types of Numbers

The fundamental number types in the Wolfram Language are represented by integers, rational, real, and complex numbers.

First, the integers have an exact result since they are numbers that cannot be represented by a decimal point.

```
In[2]:= {10, InputForm[10]}
Out[2]= {10,10}
```

Therefore, integers in the Wolfram Language are handled with infinite precision and infinite accuracy.

```
In[3]:= {10//Accuracy, InputForm[10]//Precision}
Out[3]= {∞, ∞}
```

Second, rational numbers can be represented as a quotient of two integers.

```
In[4]:= {5/10, InputForm[10/12]}
Out[4]= {1/2, 5/6}
```

Mathematica treats rational numbers exactly as with integers, so whenever Mathematica deals with rational numbers, it returns the minimum expression in which that number is represented.

```
In[5]:= {5/10 //Accuracy, InputForm[10/12] //Precision}
Out[5]= {∞, ∞}
```



Third, real numbers—typically known as floating-point numbers—are represented in the Wolfram Language by any number with a decimal point.

```
In[6]:= {2.72 //Precision, InputForm[2.72]}
Out[6]= {MachinePrecision, 2.72}
```

Since real numbers are approximate, they do not have an exact precision. These numbers are considered machine numbers, which have the precision of the `$MachinePrecision` variable. It should be noted that in the Wolfram Language, numbers 1 and 1.0 are treated differently. Although Mathematica recognizes that they are equivalent expressions, it must be taken into account that they are not the same object within the Wolfram Language.

To corroborate this, let's look at the following example, where you use `SameQ` to test if the expressions are the same for 1 and 1.0.

```
In[7]:= SameQ[1, 1.0]
Out[7]= False
```

The heads of the expressions are different because one is an integer and the other a real number.

```
In[8]:= {Head[1], Head[1.0]}
Out[8]= {Integer, Real}
```

Complex numbers are numbers that contain a real part and an imaginary part. The form of a complex number is  $a + bi$ , where “a” is the real part and “b” is the imaginary part. The symbol “i” represents the square root of the negative number -1.

```
In[9]:= 10+19I
Out[9]= 10+19I
```

The type of precision in these numbers can be exact or approximate since these numbers can be built from the numbers described previously.

```
In[10]:= {Precision[I], Precision[1 + 0.3I], FullForm[11+1I]}
Out[10]= {∞, Machineprecision, Complex[11, 1]}
```

Though complex numbers appear as a single atomic expression, these numbers can be subdivided into different expressions, such as when extracting the real or imaginary parts.

```
In[11]:= 1+I //AtomQ
```

```
Out[11]= True
```

```
In[12]:= {ReIm[1+3I],Re[1+0.3I],Im[Complex[1,0.2]]}
```

```
Out[12]= {{1,3},1.,0.2}
```

When you deal with transcendental numbers like pi and the golden ratio, these numbers are treated as symbols—that is, Mathematica has reserved these symbols since they are important numerical constants. Therefore, they have an exact precision despite being real numbers.

```
In[13]:= {Accuracy[\[Pi]],Precision[E],Accuracy[I],Precision[GoldenRatio]}
//NumberQ
```

```
Out[13]= False
```

To determine whether a given value is considered a number within the Wolfram Language, use the `NumberQ` command. It returns “True” if the expression is a number and “False” if not. This can be observed in the previous command (for transcendental numbers) and the following examples.

```
In[14]:= {NumberQ[1/2],NumberQ[1],NumberQ[E]}
```

```
Out[14]= {True,True,False}
```

As a result, you can see how a rational number and an integer are numbers, but the number `E` is not. In fact, `E` is a type of symbol.

```
In[15]:= {Head[E],FullForm[E]}
```

```
Out[15]= {Symbol,E}
```

Generally speaking, there is no restriction on combining the different types of numbers within the Wolfram Language. You can perform operations between different types.

```
In[16]:= {1+0.2+1/2+1+11+1I}
```

```
Out[16]= {13.7 +1. I}
```

Conversion between approximate numbers to exact numbers is carried out with `Rationalize`.

```
In[17]:= Rationalize[2.72]
```

```
Out[17]= 68/25
```

Also, alternative number notations like scientific notation are supported. Scientific notation is a useful tool to represent large numbers in powers of ten.

```
In[18]:= {ScientificForm[N@E/1000000], 2.71828*^-6}
Out[18]={2.71828 × 10-6, 2.71828 × 10-6}
```

You know that the `N` function is used to calculate approximate numbers. It converts an exact expression to an approximate one, keeping in mind that the desired precision can also be specified.

Different forms can generally be extrapolated to all the built-in function notations of the Wolfram Language.

- Employing the direct application of the `N` function `[ ]` to the expression

```
In[19]:= N[13/7]
Out[19]= 1.85714
```

- Utilizing the infix notation, `~N~`

```
In[20]:= E~N~E
Out[20]= 2.72
```

- Through the postfix notation, `// N`

```
In[21]:= E//N
Out[21]= 2.71828
```

- Using the prefix notation, `N@`

```
In[22]:= N@E
Out[22]= 2.71828
```

When the precision is not defined, Mathematica uses the value of `$MachinePrecision` to determine the standard precision of the approximate number. The value of `$MachinePrecision` varies since it is a float number established by Mathematica according to the characteristics of each computer.

```
In[23]:= $MachinePrecision
Out[23]= 15.9546
```

Setting arbitrary precision with SetPrecision or using machine precision.

```
In[24]:= SetPrecision[E, 17]
```

```
Out[24]= 2.7182818284590452
```

The following uses machine precision.

```
In[25]:= SetPrecision[E, MachinePrecision]
```

```
Out[25]= 2.71828
```

When precision is not introduced, Mathematica uses MachinePrecision numbers.

```
In[26]:= SetPrecision[e, MachinePrecision] == N@e
```

```
Out[26]= True
```

Another way to enter approximate numbers with some precision is by adding the grave accent symbol (‘) after the real number, followed by the precision. For example, use it for six-digit precision.

```
In[27]:= 77/3`6
```

```
Out[27]= 25.6667
```

## Working with Digits

To extract digits that make up an exact number, use the IntegerDigits function.

```
In[28]:= IntegerDigits[234544553]
```

```
Out[28]= {2,3,4,5,4,4,5,5,3}
```

RealDigits for approximate numbers.

```
In[29]:= {RealDigits[321.4546554], RealDigits[N@E]}
```

```
Out[29]= {{ {3,2,1,4,5,4,6,5,5,4,0,0,0,0,0}, 3 }, { {2,7,1,8,2,8,1,8,2,8,4,5,9,0,4,5}, 1 } }
```

In the case of a complex number, it would consist of extracting its real and imaginary parts and then extracting the digits of each part, as the case may be.

```
In[30]:= RealDigits[ReIm[113+2.7213I]]
```

```
Out[30]= {{ {1,1,3,0,0,0,0,0,0,0,0,0,0,0,0}, 3 }, { {2,7,2,1,3,0,0,0,0,0,0,0,0,0,0}, 1 } }
```

By default, the two previous functions give results in the decimal base. To define a base, enter the base you want as the function's second argument; for example, using base 2.

```
In[31]:= RealDigits[321.4546,2]
Out[31]= {{1,0,1,0,0,0,0,0,1,0,1,1,1,0,1,0,0,0,1,1,0,0,0,0,0,1,0,1,0,1,0,1,0,0,1,1,0,0,1,0,0,1,1,0,0,0,0,1,1,0,0,0,0},9}
```

Specifying the three digits of the number  $e$  in base-10 notation.

```
In[32]:= RealDigits[N@E, 10, 3]
Out[32]= {{2,7,1},1}
```

Reconstructing a number from the representation of their integers is possible with the `FromDigits` function.

```
In[33]:= FromDigits[{{2,7,1,1}}]
Out[33]= 2711
```

Also, it is possible to form a float point number.

```
In[34]:= N@FromDigits[{{2,7,1,1},1}]
Out[34]= 2.711
```

and to measure the length of an integer number.

```
In[35]:= IntegerLength[2711]
Out[35]= 4
```

## A Few Mathematical Functions

The Wolfram Language offers a wide repertoire of mathematical functions, ranging from the most basic to the most specialized. These functions can be managed numerically or symbolically, facilitating pure analytical manipulation.

Trigonometric functions are available either in radians or in degrees. Typing a number alone calculates and returns the value in radians.

```
In[36]:= Cos[Pi]
Out[36]= -1
```

Entering the number followed by the Degree unit or the symbol of degrees (°) calculates and returns the value in degrees.

```
In[37]:= Sin[90 Degree]==Sin[90\[Degree]]
Out[37]= True
In[38]:= Sin[90\[Degree]]
Out[38]= 1
```

The same applies to hyperbolic trigonometric functions and inverse trigonometric functions.

```
In[39]:= N[Cosh[Pi]]
N[Tanh[45 Degree]]
Out[39]= 11.592
Out[40]= 0.655794

In[41]:= N[ArcTan[Pi]]
N[ArcSinh[45 Degree]]
Out[41]= 1.26263
Out[42]= 0.721225
```

Logarithmic functions and exponential functions are written like common math notation. Logarithms with only a number compute the natural logarithm.

```
In[43]:= Log[E]
Out[43]= 1
```

To specify a base, type the number as the first argument and the base as the second argument.

```
In[44]:= Log[10,10]
Out[44]= 1
```

Exponentials can be written with Exp or with the constant E.

```
In[45]:= Exp[2]==E^2
Out[45]= True
```

The factorial is represented by either typing the exclamation mark after the number or by using Factorial.

```
In[46]:= 12!
```

```
Out[46]= 479001600
```

```
In[47]:= Factorial[12]
```

```
Out[47]= 479001600
```

## Numeric Function

In the Wolfram Language, functions are available for manipulating numerical data, these functions can work with any types of numbers, including real, integer, rational, and complex. Users can handle precision either exactly or using floating-point precision.

To truncate a number,  $z$ , to its closest integer ( $z$ ), use the Round function with no arguments. By adding a second argument, the Round function rounds  $z$  to the nearest multiple of the second provided number.

```
In[48]:=Round[8.9](*Rounds to 9 because it is the closest number*)
```

```
Out[48]= 9
```

```
In[49]:=Round[8.9,2](*Rounds to 8 because it is the closest multiple of  
2, 2^3*)
```

```
Out[49]= 8
```

Other similar functions that can truncate numbers given a number  $z$  are Floor and Ceiling. The Floor function rounds to the largest integer less than or equal to the number typed. The Ceiling function rounds to the smallest integer larger than or equal to the typed number.

```
In[50]:= Floor[Pi]
```

```
Out[50]= 3
```

```
In[51]:= Ceiling[Pi]
```

```
Out[51]= 4
```

The Floor and Ceiling functions can be written in their mathematical notation,  $\lfloor z \rfloor$  for Floor and  $\lceil z \rceil$  for Ceiling, by typing the key **ESC lf ESC** for the left Floor and **ESC rf ESC** for the right Floor. The same applies to Ceiling—just change **lf** for **lc** (left Ceiling) and **rf** for **rc** (right Ceiling).

```
In[52]:=  $\lfloor \text{Pi} \rfloor$ 
```

```
Out[52]= 3
```

```
In[53]:=  $\lceil \text{Pi} \rceil$ 
```

```
Out[53]= 4
```

Converting a float point number to a rational approximation can be done with Rationalize. However, adding the number 0 as the second argument can force the calculation to find the most exact form of a float point number; for example, a rational approximation to the number E.

```
In[54]:= Rationalize[N[E],0]
```

```
Out[54]= 325368125/119696244
```

The Max and Min functions return the maximum and minimum number of a list of numbers.

```
In[55]:= Max[{9,8,7,0,3,12}]
```

```
Out[55]= 12
```

```
In[56]:= Min[{0987,32,9871}]
```

```
Out[56]= 32
```

## Lists of Objects

This section extends the concept of lists in the Wolfram Language, focusing on techniques for creating and managing lists, nesting them through specialized functions, and effectively storing data in a variable. The topic covers how to create datasets and how they can be derived from various functions, as the composition of a list can include a wide range of elements, such as sets of numbers, text strings, equations, arithmetic operations, or any expression in Mathematica. Despite this, you explore concepts like arrays and sparse arrays and their respective object types. Additionally, this section discusses the nested lists and multiple ways to create data in a nested form.



## List Representation

The curly braces denote a list of general objects, with each member separated by a comma. The simplest form to create a list is to enclose data in curly braces, or by using the List function. The following examples demonstrate how to assign lists to variables and gather objects in a list.

```
In[57]:= {x2+1, "Dog",  $\pi$ }
List[1,P,Power[3,2]] (* Power[3,2] represents 3 raised to the power of 2 *)
Out[57]= {x2+1, "Dog",  $\pi$ }
Out[58]= {1,P,9}
```

The list identifier or symbol is an optional name to create the structure.

```
In[59]:= List["23.22", "Dog",  $\pi$ , 2, 4, 6, 456., 56, 2==3 && 3==2]
Out[59] = {23.22, Dog,  $\pi$ , 2, 4, 6, 456., 56, False}
```

Inside a list, between the braces, you can define all the elements that you consider suitable to be listed.

```
In[60]:= {1+I,  $\pi$  +  $\pi$ , "number 4", Sin[23 Degree], 425+I-413-3I, 24, 4456., "dog"
+ "cat"}
Out[60]= {1+I, 2 $\pi$ , number 4, Sin[23°], 12-2 I, 24, 4456., cat+dog}
```

In Mathematica, there are different types of objects. To identify an object type, you have to use the Head function. The returning value is the head of the expression, known as the data type. If you apply Head to a list, you get that the head of the expression is a list.

```
In[61]:= % //Head
Out[61]= List
```

This means that the object you have created is a List object.

## Generating Lists

Lists can be created with costume values, but Mathematica has a variety of functions to create automated lists, such as Range and Table. Both Range and Table functions create an equally spaced list of numbers. However, the Table generates a list with specified

intervals, like when “i” goes from 1 to 10. Wolfram Language also lets you incorporate built-in functions inside a list.

```
In[62]:= Range[10]
Table[i,{i,1,10}]
Table["Soccer",{i,1,15}]
Out[62]= {1,2,3,4,5,6,7,8,9,10}
Out[63]= {1,2,3,4,5,6,7,8,9,10}
Out[64]= {Soccer,Soccer,Soccer,Soccer,Soccer,Soccer,Soccer,Soccer,Soccer,
Soccer,Soccer,Soccer,Soccer,Soccer,Soccer}
```

The Table function can also be used to create indexed lists. Each interval is specified within the curly braces {}, as shown in the previous and following examples.

```
In[65]:= Table["Red and Blue",5]
Range[-5,5]
Out[65]= {Red and Blue,Red and Blue,Red and Blue,Red and Blue,Red and Blue}
Out[66]= {-5,-4,-3,-2,-1,0,1,2,3,4,5}
```

The Table function can work with or without an inner iterator, but to create structured lists, using an iterator is recommended.

```
In[67]:= Table[i^i,{i,1,5}]
Out[67]= {1,4,27,256,3125}
```

This shows the function without an iterator.

```
In[68]:= Table[10^3,{5}]
Out[68]= {1000,1000,1000,1000,1000}
```

---

**Note** When using the iterator, make sure to properly write the expression to avoid errors. When the table recognizes the iterator, it changes colors because the letter is no longer a symbol.

---

You can create a list of lists. This type of structure is considered a nested list.

```
In[69]:= {Range[5], Table[h, {h, -6, 2}]}
Out[69]= {{1, 2, 3, 4, 5}, {-6, -5, -4, -3, -2, -1, 0, 1, 2}}
```

The iterator can also be an alphanumeric variable.

```
In[70]:= Table[data2, {data2, 0, 6}]
Out[70]= {0, 1, 2, 3, 4, 5, 6}
```

Structures of arrays with the same data can also be created, such as an array of 2×2.

```
In[71]:= Table[11, {2}, {2}]
Out[71]= {{11, 11}, {11, 11}}
```

The Table function supports multiple iterators, which is useful when constructing tabular data.

```
In[72]:= Table[i+j+k, {i, 1, 4}, {j, 1, 4}, {k, 1, 4}]
Out[72]= {{ {3, 4, 5, 6}, {4, 5, 6, 7}, {5, 6, 7, 8}, {6, 7, 8, 9}}, { {4, 5, 6, 7}, {5, 6, 7, 8}, {6, 7, 8, 9}, {7, 8, 9, 10}}, { {5, 6, 7, 8}, {6, 7, 8, 9}, {7, 8, 9, 10}, {8, 9, 10, 11}}, { {6, 7, 8, 9}, {7, 8, 9, 10}, {8, 9, 10, 11}, {9, 10, 11, 12}}}
```

To display a list in a more structured way using the Grid command.

```
In[73]:= Table[i-j, {i, 1, 2}, {j, 1, 2}]//Grid
Out[73]= 0    -1
          1    0
```

An alternative to the Grid command is the TableForm command, which lets you display the list created as a table. This command is explained in detail later.

```
In[74]:= Table[i+j, {i, 1, 2}, {j, 4, 6}]//TableForm
Out[74]//TableForm= 5    6    7
                     6    7    8
```

There is no limitation on the intervals of the iterators. You can choose that “i” goes from 0 to 3 and “j” from “i” to 3 and use TableForm to view it.

```
In[75]:= Table[{i, j}, {i, 3}, {j, i, 3}]//TableForm
Out[75]//TableForm= 1 1 1
                    1 2 3
                    2 2
                    2 3
                    3
                    3
```

You can even use other syntax notations like the increment (++) or decrement (--) in the interval iterator.

```
In[76]:= Table[{i,j},{i,2},{j,i++,2}]
Out[76]= {{2,1},{2,2}},{3,2}}
```

The increment (++) and decrement (--) operators can also be used in assigned variables; this operator can have precedence or posteriority. When written before the variable, they are called PreIncrement or PreDecrement.

```
In[77]:= x=0;x++;x (*applied on the current value and shown next time x is
called*)
Out[77]= 1
```

```
In[78]:= Clear[x];x=0;--x (*applied on the current value and shown when x
is called*)
Out[78]= -1
```

Alternatively, you can apply replacement rules with the symbol (/.). For example, you create a list of random integers consisting of 0s or 1s, then replace the 1s with 2s whenever they appear. Add a space between the condition expressions to avoid a typo error and the correct right arrow (\[Rule]). Another form of Table can also be used with explicit values for the iterator.

```
In[79]:= Table[RandomInteger[],{i,1,10}]/. 1->2
Out[79]= {2,0,2,0,2,2,0,0,2,2}
```

```
In[80]:= Table[i^2,{i,{1,2,3,4,5}}]
Out[80]= {1,4,9,16,25}
```

## Arrays of Data

There are different forms to create an array. The most used form is a list, as you saw in the previous section. But as an alternative to the Table command or Range command, arrays can be created with the Array command, which generates a list with a specific function applied to the elements created. The Array, ConstantArray, and SparseArray functions can also be used to build lists. The form of these functions is analogous to the previous ones.

```
In[81]:= Array[Cos[90 Degree],{3,3}]/Grid
Out[81]= 0[1,1]    0[1,2]    0[1,3]
          0[2,1]    0[2,2]    0[2,3]
          0[3,1]    0[3,2]    0[3,3]
```

What happens with Array is that it constructs an array from a function. In the previous example, you generated an array from the numerical value of the cosine of 90 degrees, followed by the structure of the array, which is 3×3. The indices on the right side of the array values are the positions of each element in the array.

If you generalize to any function, you can better see how Array works.

```
In[82]:= Array[F,{2,2}]/Grid
Out[82]= F[1,1]    F[1,2]
          F[2,1]    F[2,2]
```

As you can observe, the F function is applied and is respective to each element of the arrangement.

To create an array of constant values the ConstantArray function is used. To write the function, first write the value you want to repeat, followed by the times you want it to repeat.

```
In[83]:= ConstantArray[\[Pi],5]
Out[83]= {π,π,π,π,π}
```

You can also create arrangements with defined dimensions.

```
In[84]:= ConstantArray[\[Pi],{4,4}]
Out[84]= {{π,π,π,π},{π,π,π,π},{π,π,π,π},{π,π,π,π}}
```

To display a data array, there is the MatrixForm command, which, as its name suggests, shows the array in matrix form.

```
In[85]:= ConstantArray[\[Pi],{4,4}]/MatrixForm
```

```
Out[85]/MatrixForm= 
$$\begin{pmatrix} \pi & \pi & \pi & \pi \\ \pi & \pi & \pi & \pi \\ \pi & \pi & \pi & \pi \\ \pi & \pi & \pi & \pi \end{pmatrix}$$

```

A sparse arrangement is one in which the elements generally have the same value. The SparseArray command lets you define the values of the array positions. By standard, if any position is not defined, the value is 0.

The SparseArray command generates an object of type SparseArray, shown in Figure 2-2, with the name of the command and a gray box that appears.

```
In[86]:= SparseArray[{{1,1},{2,2}}->{1,2}]
Out[86]=
```

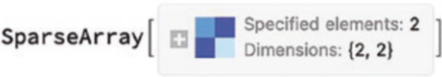


Figure 2-2. SparseArray object

If you click the + icon, you see the array’s characteristics and its rules; this is shown in Figure 2-3.

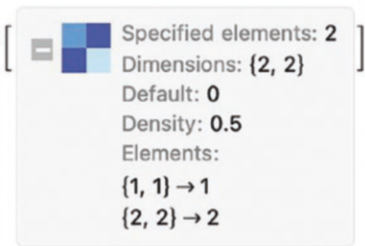


Figure 2-3. Specifications of the array

In the Wolfram Language, there is no limitation on the content of a SparseArray. Furthermore, you can create an array with the same values on its diagonal.

Figure 2-4 illustrates elements of the same values in the array appear in one color, and different values appear in another.

```
In[87]:= SpArray=SparseArray[{{1,1}->"A",{2,2}->"A",{3,3}->"A",{4,4}->"A"},{4,4}]
Out[87]=
```

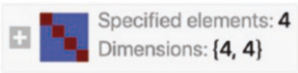


Figure 2-4. Sparse Array with more elements

With the help of `MatrixForm`, you can visualize the arrangement as a matrix.

```
In[88]:= MatrixForm[%]
```

```
Out[88]//MatrixForm=
```

$$\begin{pmatrix} A & 0 & 0 & 0 \\ 0 & A & 0 & 0 \\ 0 & 0 & A & 0 \\ 0 & 0 & 0 & A \end{pmatrix}$$

To convert the sparse array object to a list object, use `Normal` to normalize into expression form.

```
In[89]:= Normal[SpArray]
```

```
Out[89]= {{A,0,0,0},{0,A,0,0},{0,0,A,0},{0,0,0,A}}
```

And now you deal with a list.

```
In[90]:= Head[%]
```

```
Out[90]= List
```

## Nested Lists

A nested list is a list of lists where the elements of the lists correspond to another list, and so on. Nested lists can be used for ordered or unordered data structures. To create a nested list, you can use curly braces within curly braces or built-in functions.

```
In[91]:= {"This","is","A"}, {"Nested","List","."}
```

```
Out[91]= {{This,is,A},{Nested,List,.}}
```

You can also use the `Table` function.

```
In[92]:= Table[Prime[i]+Prime[j],{i,1,3},{j,2,4}]
```

```
Out[92]= {{5,7,9},{6,8,10},{8,10,12}}
```

To measure a list, you must use the `Length` command.

```
In[93]:= NestL=Table[Prime[i]+RandomReal[j],{i,1,3},{j,1,3}];
```

```
Length[NestL]
```

```
Out[93]= 3
```

The length of the list is 3 because `Length` is properly used with flattened lists. To properly measure the depth of a nested list, `Dimensions` is more suited for the task.

```
In[94]:= Dimensions[NestL]
Out[94]= {3,3}
```

`Dimensions` provide a general aspect of the dimensions of the nested list, meaning that a list of three sublists constitutes your list and that the sublists each have three elements. Mathematica constructs a list with three elements, in which those three elements are also a list, and those lists have three elements, and each element corresponds to a specific value form.

---

**Note** You might want to use `TreeForm` to explore how Mathematica deals with nested list expressions; for instance, `(*TreeForm[NestL]*)`.

---

The `ArrayDepth` command measures the depth of a nested list or an array.

```
In[95]:= ArrayDepth[NestL]
Out[95]= 2
```

Now you know programmatically that `NestL` has a depth of 2.

## Vectors

Mathematica handles vectors the same way as with lists. Usual calculations of linear algebra can be symbolic or numeric.

```
In[96]:= V={6,3,2}
Out[96]= {6,3,2}
```

A vector is always shown as a list. To see a vector in regular notation, the `MatrixForm` command is used.

```
In[97]:= MatrixForm[V]
Out[97]//MatrixForm=
```

$$\begin{pmatrix} 6 \\ 3 \\ 2 \end{pmatrix}$$



The VectorQ command can tell you if the list you are dealing with is a vector.

```
In[98]:= VectorQ[V]
```

```
Out[98]= True
```

To see the rank of the vector, use either ArrayDepth or TensorRank.

```
In[99]:= {TensorRank[V], ArrayDepth[V]}
```

```
Out[99]= {1, 1}
```

Vectors are created with the same commands that create a list: Table, Array, Range, curly braces, SparseArray, ConstantArray, and so forth. Also, common operations of vectors are performed like normal lists.

```
In[100]:=
```

```
Print["Addition: "<>ToString[V+V]]
```

```
Print["Subtraction: "<>ToString[V-V]]
```

```
Print["Scalar product: "<>ToString[2*V]]
```

```
Print["Cross product: "<>ToString[Cross[V,{1,3,2}]]]
```

```
Print["Norm: "<>ToString[Norm[V]]]
```

```
Addition: {12, 6, 4}
```

```
Subtraction: {0, 0, 0}
```

```
Scalar product: {12, 6, 4}
```

```
Cross product: {0, -10, 15}
```

```
Norm: 7
```

## Matrixes

A matrix is a square list or list of lists arranged in n-rows and m-columns, where n and m are the dimensions of the matrix.

$$A_{m \times n} = \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{pmatrix}$$

The easiest form is to create a list of lists.

```
In[105]:= {{3,3,1},{7,8,7}}//MatrixForm
```

```
Out[105]//MatrixForm=
```

$$\begin{pmatrix} 3 & 3 & 1 \\ 7 & 8 & 7 \end{pmatrix}$$

Another way is to go to Insert ► Table/Matrix ► New. A pop-up menu appears; select Matrix and specify the rows and columns within this menu. With this option, you can also specify to fill contents and the diagonal and add a grid or frames, such as in the next example that has drawn lines between columns.

$$\text{In[106]} := A = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

```
Out[106]:= {{1,0,0},{0,1,0},{0,0,1}}
```

To test whether a list of lists is a matrix, use MatrixQ.

```
In[107]:= MatrixQ[A]
```

```
Out[107]= True
```

Transpose returns the transpose of a matrix—that is, changing its rows by columns. For matrix **A**, the transpose is denoted by  $A^T$ .

```
In[108]:= Transpose[{{0,1,0},{0,1,0},{0,1,0}}]//MatrixForm
```

```
Out[108]//MatrixForm=
```

$$\begin{pmatrix} 0 & 0 & 0 \\ 1 & 1 & 1 \\ 0 & 0 & 0 \end{pmatrix}$$

## Matrix Operations

Common operations between matrixes are performed by the rules of linear algebra: addition, subtraction, and multiplication. Remember that when multiplying two matrixes, **A** and **B**, the number of columns in **A** must match the number of rows in **B**. In mathematical terms:  $A_{m \times n} \times B_{n \times l} = C_{m \times l}$ .

```
In[109]:= B={{0,1,0},{0,1,0},{0,1,0}};
Print["Addition: "<>ToString[A+B]]
Print["Subtraction: "<>ToString[A-B]]
Print["Product: "<>ToString[Dot[B,V]]]
Addition: {{1, 1, 0}, {0, 2, 0}, {0, 1, 1}}
Subtraction: {{1, -1, 0}, {0, 0, 0}, {0, -1, 1}}
Product: {3, 3, 3}
```

To calculate the determinant, use Det.

```
In[113]:= {Det[A],Det[B]}
Out[113]= {1,0}
```

To construct a diagonal matrix, use the DiagonalMatrix command; for the identify matrix, use the IdentityMatrix command. DiagonalMatrix is for costume values, and IdentityMatrix returns a matrix with a diagonal with the same elements.

```
In[114]:= DiagonalMatrix[{X,Y,Z}]/MatrixForm
IdentityMatrix[{2,2}]/MatrixForm(*Identity matrix of 2 by 2*)
Out[114]/MatrixForm=
```

$$\begin{pmatrix} X & 0 & 0 \\ 0 & Y & 0 \\ 0 & 0 & Z \end{pmatrix}$$

```
Out[115]/MatrixForm=
```

$$\begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$$

## Restructuring a Matrix

Matrix restructuring is done with the same commands to restructure a list, like replacing an element with a new value.

```
In[116]:= ReplacePart[A,{{1,1},{2,2}}-> 3]//MatrixForm
Out[116]//MatrixForm=
```

$$\begin{pmatrix} 3 & 0 & 0 \\ 0 & 3 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

Also, it can be done by assigning the value. To access the elements of a matrix, enter the symbol followed by the subscript of the element of interest with the double bracket notation ([ ]). Later, you see the proper functionality of this short notation. In this case, you change the value of the element in position 1,1 of the matrix.

```
In[117]:= A[[1,1]] = 2;
MatrixForm[A]
Out[118]//MatrixForm=
```

$$\begin{pmatrix} 2 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

If matrix A is called again, the new value is preserved. To invert a square matrix, use Inverse.

```
In[119]:= Inverse[A]//MatrixForm
Out[119]//MatrixForm=
```

$$\begin{pmatrix} 1/2 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

Measuring the dimensions of a matrix is done by using Dimensions.

```
In[120]:= Dimensions[A]
Out[120]= {3,3}
```

# Manipulating Lists

The previous section demonstrated different ways to create lists, including arrays, nested lists, and tables. This section describes how to manipulate these lists through referenced names, functions, and compact notation. You learn how to access the data of a list depending on its position. You learn how to add and delete elements in a list, replace single parts, and change the value of a specific element. You also examine restructuring lists once it has been built, ordering them, and converting nested lists to linear lists based on their depth. Finally, the section investigates how to see data from a list through patterns and examine pattern behavior in the Wolfram Language.

## Retrieving Data

Several functions exist for handling elements of a list. The Part ["list", i] function allows you to select index parts of a list with index i.

---

**Note** The index in a list starts at 1. Index 0 is for the head of the list.

---

For example, let you define a list called list1 and use Part to access the elements inside the list. The Part function works by defining the position of the element you want.

```
In[121]:= list1={1,2};
Part[{1,2},1]
Out[122]= 1
```

It also works with index notation.

```
In[123]:= {1,2}[[1]]
Out[123]= 1
```

Lists can be fully referenced by using their assigned names. Elements inside the structure can be accessed using the notation of double square brackets [[ i ]] or with the special character notation of double brackets, `[[ ]]`.

---

**Tip** To introduce the double square bracket character, type Esc [[ Esc and Esc ]] Esc.

---

```
In[124]:= list1[[1]] (*[[i]] gives you access to the element of the list in
the position i.*)
```

```
Out[124]= 1
```

---

**Note** Square brackets ([ [ ] ]) are the short notation for part Esc.

---

To access the elements of the list by specifying the positions, you can use the span notation, which is with a double semicolon (;;).

```
In[125]:= list2=List[34,6,77,4,5,6];
Part[list2,1;;4] (*from items 1 to 4*)
Out[126]= {34,6,77,4}
```

You can also use backward indices, where the counts start from right to left, which is from the last element to the first. Let you now select from position -6 to -4.

```
In[127]:= list2[[-6;;-4]]
Out[127]= {34,6,77}
```

For the nested list, the same process is applied. The concept can be extended into a more general aspect. The next example creates a nested list with three levels and select a unique element.

```
In[128]:= list3=List[2^3,2.72,{\[Beta],ex,{Total[1+2],"Plane"}}];
list3[[3,3,2]]
Out[129]= Plane
```

In the previous example, you created a nested list of depth three. Next, you select the third element of the list {8, 2.72, { $\beta$ , ex, {Total[1 + 2], "Plane"}}}, then from that list, select the third element of the previous list, which is {Total[1 + 2], "Plane"}. Finally, you select the element in the second position of the last list, which is "Plane".

If you are dealing with a nested list, use the same concept you saw with the span notation. The next example selects the third element of the list3 and then display from position 1 to 2.

```
In[130]:= list3[[3,1;;2]]
Out[130]= { $\beta$ , ex}
```

The same is done to a more in-depth list; you use the list's third element, then display from position 3 to 3 and select part 1.

```
In[131]:= list3[[3,3;;3,1]]
Out[131]= {3}
```

Segments of data can be displayed based on what parts of the data you are interested in. For example, the `Rest` function shows the data elements, except for the first. `Most` display the whole list except for the last element(s), depending on the type of list.

```
In[132]:= Rest[list3]
Out[132]= {2.72, { β, ex, {3, Plane}}}
```

```
In[133]:= Most[list3]
Out[133]= {8, 2.72}
```

An alternative to the previous functions is the `Take` function, which lets you select more broadly the data in a list. There are three possible ways to accomplish this.

- By specifying the first *i* elements

```
In[134]:= Take[list3, 2]
Out[134]= {8, 2.72}
```

- By specifying the last *-i* elements

```
In[135]:= Take[list3, -1]
Out[135]= {{ β, ex, {3, Plane}}}
```

- By selecting the elements from *i* to *j*

```
In[136]:= Take[list3, {1, 3}]
Out[136]= {8, 2.72, { β, ex, {3, Plane}}}
```

## Assigning or Removing Values

Once a list is established—if you have defined a name for it—it can be used just like any other type. This means that elements can be replaced by others. To change a value or values, select the position of the item, and then set the new value.

```
In[137]:= list4={"Soccer","Basketball",0,9};
list4[[2]]=1 (*position 2 corresponds to the string Basketball and we
change it for the number 1*)
Out[138]= 1
```

You can check that the new values have been added.

```
In[139]:= list4
Out[139]= {Soccer,1,0,9}
```

In addition to using the abbreviated abbreviation notation, you can use the Replace function part of specific values and choose the list, the new element, and the position.

```
In[140]:= ReplacePart[list4,Exp[X],4]
Out[140]= Soccer, 1, 0,  $e^x$ 
```

To add new values, use PrependTo and AppendTo; the first adds the value on the left side of the list, whereas the second adds it by the right side of the list. Append and Prepend operate the same but with storing the new value in the original variable.

```
In[141]:= PrependTo[list4,"Blue"]
Out[141]= {Blue,Soccer,1,0,9}
```

```
In[142]:= AppendTo[list4,4]
Out[142]= {Blue,Soccer,1,0,9,4}
```

```
In[143]:= list4(*we can check the addition of new values.*)
Out[143]= {Blue,Soccer,1,0,9,4}
```

To remove the values of the list, you use Drop. Drop can work with the level of the specification or the number of elements to be erased.

```
In[144]:= Drop[list4,3];(*first 3 elements,Delete[list3,3]*)
Drop[list4,{5}](or by position,position, number 5*)
Out[145]:= {Blue,Soccer,1,0,4}
```

The Delete command can also do the job by defining the particular positions on the list—for example, deleting the contents in positions 1 and 5.

```
In[146]:= Delete[list4,{1},{5}]
Out[146]= {Soccer,1,0,4}
```



As an alternative to Append and Prepend, there is the Insert function, with which you can add elements indicating the position where you want the new data. Given the expression (list4), insert the new element (2/43.23) at the third position. Consequently, the number 2/43.23 now occupies the list's third slot.

```
In[147]:= Insert[list4,2/43.23,3]
Out[147]= {Blue,Soccer,0.0462642,1,0,9,4}
```

The Insert function allows the use of several positions at the same time; for example, inserting the number 0.023 at positions -6 (second) and 7 (the last position).

```
In[148]:= Insert[list4,0.023,{{-6},{7}}]
Out[148]= {Blue,0.023,Soccer,1,0,9,4,0.023}
```

If you want to add repetitive terms or remove terms to a list or an array, you can use the ArrayPad function. The standard value is zeros if the term to be added is not defined.

```
In[149]:= ArrayPad[list4,1>(*number 1 means one zero each side*)
Out[149]= {0,Blue,Soccer,1,0,9,4,0}
```

If you want to add one-sided terms, it is written as follows.

```
In[150]:= ArrayPad[list4,{1,2}>(*1 zero to the left and 2 zeros to
the right*)
Out[150]= {0,Blue,Soccer,1,0,9,4,0,0}
```

To add values other than zero, you must write the value to the right of the number of times the value is repeated.

```
In[151]:= ArrayPad[list4,{0,3},"z"](*Adding the letter z three times only
the right side*)
Out[151]= {Blue,Soccer,1,0,9,4,z,z,z}
```

With ArrayPad you can add reference lists; for example, add a new list of values either left or right.

```
In[152]:= newVal={0,1,4,9}; (*Here we add them on the left side*)
ArrayPad[list4,{4,0},newVal]
Out[153]= {4,9,0,1,Blue,Soccer,1,0,9,4}
```

ArrayPad also can remove elements from a list symmetrically using negative indices.

```
In[154]:= ArrayPad[list4,-1>(*it deletes the first and last elements*)
Out[154]= {Soccer,1,0,9}
```

---

**Note** With ArrayPad, addition and deletion are symmetric unless otherwise specified.

---

## Structuring List

When you work with lists, in addition to the different forms of access and removing its content, you might encounter cases where a list needs to be accommodated, sectioned, or restricted. The following explores several forms to achieve these tasks.

To sort a list into a specific order, use Sort followed by the sorting function.

```
In[155]:= Sort[{1,12,2,43,24,553,65,3},Greater]
Out[155]= {553,65,43,24,12,3,2,1}
```

Sort by default sorts values from less to greater, either numbers or text.

```
In[156]:= Sort[{"b","c","zz","sa","t","p"}]
Out[156]= {b,c,p,sa,t,zz}
```

To reverse a list, use the Reverse command.

```
In[157]:= Reverse[{1,12,2,43,24,553,65,3}]
Out[157]= {3,65,553,24,43,2,12,1}
```

To create a nested list in addition to that previously seen, you can generate partitions to a flat list by rearranging the elements of the list. For example, you create partitions of a list to subdivide the list into pairs.

```
In[158]:= Partition[{1,12,2,43,24,553,65,3},2]
Out[158]= {{1,12},{2,43},{24,553},{65,3}}
```

You can choose a partition with successive elements included.

```
In[159]:= Partition[{1,12,2,43,24,553},3,1]
Out[159]= {{1,12,2},{12,2,43},{2,43,24},{43,24,553}}
```

Depending on how you want a nested list, you can add an offset to the partition; for example, a partition in two with an offset of four.

```
In[160]:= Partition[{"b","c","zz","sa","t","p"},2,4]
Out[160]= {{b,c},{t,p}}
```

To return to a flat list, the Flatten function is used.

```
In[161]:= Flatten[{{1,12},{2,43},{24,553},{65,3}}]
Out[161]= {1,12,2,43,24,553,65,3}
```

Depending on the depth of the list, you can decide how deep the Flatten should be.

```
In[162]:= Flatten[{{{1},1},1},1] (*here we flatten a list with a level
1 depth.*)
Out[162]= {{{1},1},1,1}
```

The ArrayReshape function lets you reshape data into a specific rectangular array with; for example, create an array of 3×3.

```
In[163]:= ArrayReshape[{1,12,2,43,24,553,65,3},{3,3}]
Out[163]= {{1,12,2},{43,24,553},{65,3,0}}
```

Elements that complete the array form are zeros. This is shown in the next example using ArrayShape to create an array of 2×2 from one element in the list.

```
In[164]:= ArrayReshape[{6},{2,2}]
Out[164]= {{6,0},{0,0}}
```

When dealing with a nested list, SortBy is also used, but instead of a sorting function, a built-in function is used. For example, order a list by the result of their approximate value.

```
In[165]:= SortBy[{1,4,553,12.52,4.3,24,7/11},N]
Out[165]= {7/11,1,4,4.3,12.52,24,553}
```

## Criteria Selection

Particular values of a list can be selected with certain conditions; conditions can be applied to lists by using the `Select` command. The function selects the elements of the list that are true to the criteria established; the functions used for criteria can be order functions.

```
In[166]:= nmbrList=List[12,5,6,345,7,3,1,5];
Select[nmbrList,EvenQ] (*only the values that return True are selected, in
this case values that are even*)
Out[167]= {12,6}
```

`Pick` is also an alternative to `Select`.

```
In[168]:= Pick[nmbrList,PrimeQ @ nmbrList]
Out[168]= {5,7,3,5}
```

Pattern matching is used in the Wolfram Language to decree whether a given criterion should be associated with an expression. In the context of the Wolfram Language, three distinct types of patterns exist.

- The underscore symbol (`_`) represents any expression within the Wolfram Language.
- The double underscore symbol (`__`) represents a sequence containing one or more expressions.
- The triple underscore symbol (`___`) represents a sequence containing zero or more expressions.

Every pattern has its built-in function name. One underscore is `Blank`, two underscores are `BlankSequence`, and three underscores are `BlankNullSequence`.

To better understand the following examples in the channels, you use the `Cases` function, which allows you to select data that corresponds to the pattern.

The following is a list of data pairs where you write the selection pattern (`_`).

```
In[169]:= Cases[{ {1,1}, {1,2}, {2,1}, {2,2} }, {_}]
Out[169]= {}
```

It does not choose any element because it does not have the form of the list pattern; for example, the form `{a,b}`. Now if you change this shape, you see that it selects all the elements that match the shape of the pattern.

```
In[170]:= Cases[{{1,1},{1,2},{2,1},{2,2}},{_,_}]
Out[170]= {{1,1},{1,2},{2,1},{2,2}}
```

The same result can be obtained if you use the double underscore.

```
In[171]:= Cases[{{1,1},{1,2},{2,1},{2,2}},{__}]
Out[171]= {{1,1},{1,2},{2,1},{2,2}}
```

The following example shows how to select data from a list that contains numerical and categorical data. You use the `RandomChoice` function, which gives you a random selection from a list. In this case, it is a random selection between the words `Red` or `Blue`. The next chapter explains how this random function works in the Wolfram Language.

```
In[172]:= SeedRandom[1234]; (*Employ SeedRandom[s] to ensure the same
sequence of pseudorandom in the following examples.*)
tbl=Table[{i,j,k,RandomChoice[{"Red","Blue"}]},{i,1,3},{j,1,3},{k,1,3}]/
TableForm
Out[173]//TableForm=
```

1 1 1 Blue	1 2 1 Red	1 3 1 Red
1 1 2 Blue	1 2 2 Red	1 3 2 Red
1 1 3 Blue	1 2 3 Red	1 3 3 Red
2 1 1 Blue	2 2 1 Blue	2 3 1 Blue
2 1 2 Blue	2 2 2 Red	2 3 2 Red
2 1 3 Red	2 2 3 Red	2 3 3 Blue
3 1 1 Blue	3 2 1 Red	3 3 1 Red
3 1 2 Red	3 2 2 Blue	3 3 2 Red
3 1 3 Blue	3 2 3 Red	3 3 3 Red

The numbers on the right side are named `Red` or `Blue`. For example, you can use `Cases` to choose the values in the `Blue` or `Red` category. Since this is a nested list of depth four, you must specify the level (`{4}`) at which `Cases` should search for patterns.

```
In[174]:= Cases[tbl,{_,_,_, "Blue"},{4}]
Out[174]=
{{1,1,1,Blue},{1,1,2,Blue},
{1,1,3,Blue},{2,1,1,Blue},{2,1,2,Blue},
{2,2,1,Blue},{2,3,1,Blue},{2,3,3,Blue},
{3,1,1,Blue},{3,1,3,Blue},{3,2,2,Blue}}
```

Furthermore, the same result can be obtained using the double underscore. Using only the number 4, search in levels from 1 through 4.

```
In[175]:= Cases[tbl,{__,"Blue"},{4}]
Out[175]=
{{1,1,1,Blue},{1,1,2,Blue},
{1,1,3,Blue},{2,1,1,Blue},{2,1,2,Blue},
{2,2,1,Blue},{2,3,1,Blue},{2,3,3,Blue},
{3,1,1,Blue},{3,1,3,Blue},{3,2,2,Blue}}
```

You can even count how much of the Blue category you have.

```
In[176]:= Count[Tbl,{__,"Blue"},{4}]
Out[176]= 11
```

Count works in the next form, Count["list", pattern, level of spec].

Now that you understand the underscore function, you can use the Cases function to check conditions and filter values. To attach a condition, use the form (/; "condition"), where the symbol /; followed by a rule or pattern indicates that the subsequent expression is a condition or pattern in Mathematica. In the next example, the  $x_$  represents an arbitrary element  $x$ , which represents the list's elements in this case. The condition that  $x$  is greater than 5 is then applied.

```
In[177]:= Cases[nmbrList,x_ /;x>5]
(*only the values greater than 5 are selected.*)
(*x can be replaced by any arbitrary symbol try using z_ and z > 5, the
result should be the same *)
Out[177]= {12,6,345,7}
```

As you saw in the previous example, what happens when you use  $_$  means that the expression  $x_$  must be applied to the condition  $> 5$  since  $_$  means any expression, which is the list.

Cases can also select data where the condition is true for the established pattern or set of rules. The next example selects data that are integers. The pattern objects are represented by an underscore or a rule of expression.

```
In[178]:=mixList={1.,1.2,"4",\[Pi],{"5.2","Dog"}, 3,66,{Orange,Red}};
Cases[mixList,_Integer]
(*We now select the numbers that are integers*)
Out[179]= {3,66}
```

The underscore can be applied to patterns that check the head of an expression, which is an integer. Cases compare each element to see if they are integers.

As for conditional matching, if the blanks of a pattern are accompanied by a question mark (?) and then the function test, the output is a Boolean value.

```
In[180]:= MatchQ[mixList,_?ListQ](*we test if mixlist has a head of List*)
Out[180]= True
```

You can select the level of specification with Cases. The next example selects the cases that are a string; you write two as a level of specification because mixList is a nested list with two sublists.

```
In[181]:= Cases[mixList,_?StringQ,2]
Out[181]= {4,5.2,Dog}
```

You can include several patterns with alternatives. To test different alternatives, place a (|) between patterns, so it resembles the form “pattern1” | “pattern2” | “pattern3” | ...

```
In[182]:= Cases[mixList, _?NumberQ| _?String] (*We select the numbers and
the strings*)
Out[182]= {1.,1.2,3,66}
```

## Summary

This chapter serves as an opening to the concept of lists, which are a core structure employed in Mathematica. It emphasizes the utility of lists and presents the unique Wolfram Language syntax. The chapter covers diverse types of objects that can be represented as lists. It concludes with basic functionalities for manipulating lists based on data requirements.

## CHAPTER 3

# Working with Data and Datasets

This chapter reviews the basics of working with data and datasets in the Wolfram Language. It starts by reviewing how to apply functions to a list, followed by how to define user functions that can be used throughout a notebook. Next, you are introduced to how to write code in one of the powerful syntaxes used in the Wolfram Language, called pure functions. Naturally, you then delve into associations, explaining how to associate keys with values and why they are fundamental for proper dataset construction in the Wolfram Language. The chapter concludes with an overview of how associations are abstract constructions of hierarchical data representations.

## Operations with Lists

Let's look at how to perform operations on and between lists. This is important since, for the most part, results in Mathematica can be treated as lists. This section explains how to perform arithmetic operations, addition, subtraction, multiplication, division, and scalar multiplication. You also learn how to apply functions to a list using Map and Apply. These tools are helpful when dealing with linear and nested lists because they allow you to specify a function's depth level of application. This section also discusses how to make user-defined functions, their syntax, term grouping, receive groups, and apply the function like any other. It reviews an important concept of the Wolfram Language, which is pure functions, since these are very important for carrying out powerful tasks and activities and compactly writing code.



## Arithmetic Operations to a List

This section discusses how lists support different arithmetic operations between numbers and between lists. You can perform basic arithmetic operations like addition, subtraction, multiplication, and division with lists.

### Addition and Subtraction

The following are examples of addition and subtraction operations.

```
In[1]:= List[1,2,3,4,5,6]+1
Out[1]= {2,3,4,5,6,7}
In[2]:= List[1,2,3,4,5,6]-5
Out[2]= {-4,-3,-2,-1,0,1}
```

### Division and Multiplication

The following are examples of division and multiplication operations.

```
In[3]:= List[1,2,3,4,5,6]/ $\pi$ 
Out[3]=  $\left\{\frac{1}{\pi}, \frac{2}{\pi}, \frac{3}{\pi}, \frac{4}{\pi}, \frac{5}{\pi}, \frac{6}{\pi}\right\}$ 
```

Scalar multiplication operations can also be performed.

```
In[4]:= List[1,2,3,4,5,6]*2
Out[4]= {2,4,6,8,10,12}
```

### Exponentiation

The following is an example using exponentiation.

```
In[5]:= List[1,2,3,4,5,6]^3
Out[5]= {1,8,27,64,125,216}
```

Lists can also support basic arithmetic operations between lists.

```
In[6]:= List[1,2,4,5]-List[2,3,5,6]
Out[6]= {-1,-1,-1,-1}
```

You can also use mathematical notation to perform operations.

```
In[7]:= {"Dog",2}
        {2,1}
```

```
Out[7]= {Dog,2}
        2
```

To perform computations between lists, the length of the lists must be the same; otherwise, Mathematica returns an error specifying that lists do not have the same dimensions, like in the following example.

```
In[8]:= {1,3,-1}+{-1}
```

```
During evaluation of In[8]:= Thread::tlen: Objects of unequal length in
{1,3,-1}+{-1} cannot be combined.
```

```
Out[8]= {-1}+{1,3,-1}
```

## Joining a List

To join one list with another—that is, to join the two lists—there is the `Union` command, which joins the elements of the lists and shows it as a new list.

```
In[9]:= Union[List["1","v","c"],{13,4,32},List["adfs",3,1,"no"]]
```

```
Out[9]= {1,3,4,13,32,1,adfs,c,no,v}
```

In addition to the `Union` command, there is the `Intersection` command, which has a function analogous to what it represents in set theory. This command lets you observe the common elements in the list or lists.

```
In[10]:= Intersection[{7,4,6,8,4,7,32,2},{123,34,6,8,5445,8}]
```

```
Out[10]= {6,8}
```

As seen the lists only have in common the numbers 6 and 8.

## Applying Functions to a List

Functions can be concisely applied and automated to a list. The most used functions are `Map` and `Apply`. A short notation is to use the symbol `@` instead of the square brackets `[ ]`; `f@“expr”` is equivalent to `f[expr]`.

```
In[11]:= Max@{1,245.2,2,5,3,5,6.0,35.3}
Out[11]= 245.2
```

Map has the following form, Map[f, “expr”]; another way of showing it is with the shorthand notation using the symbol @. f /@ “expr” and Map[f, “expr”] are equivalent. This function also supports nested lists.

```
In[12]:= Factorial/@List[1,2,3,4,5,6]
Out[12]= {1,2,6,24,120,720}
```

Map can be applied to nested lists.

```
In[13]:= Map[Sqrt,{{1,2},{3,4}}]
Out[13]= {{1,Sqrt[2]},{Sqrt[3],2}}
```

The Map function is applied to each element of the list. Map can also work with nested lists, as in the previous example. The next example creates a list of 10 elements with Table. Those elements are random numbers between 0 and 1, and then you map a function to convert them to string expressions.

```
In[14]:= data=Range[RandomReal[{0,1}],10];(*List*)
ToString/@data (*mapping a to convert to string*)
Head/@% (*Checking the data type of every element*)
Out[15]= {0.526418,1.52642,2.52642,3.52642,4.52642,5.52642,6.52642,7.52642,
8.52642,9.52642}
Out[16]= {String,String,String,String,String,String,String,String,String,
String}
```

Let’s look at how to apply a function to a list with additional functions. Apply has the form Apply [f, “expr”] and the shorthand notation is f @@ “expr”.

```
In[17]:= Apply[Plus,data](*It gives the sum of the elements of Data*)
Out[17]= 50.2642
```

```
In[18]:= Plus@@data
Out[18]= 50.2642
```

Also, commands can be applied to a list in the same line of code, which is helpful when dealing with large lists. For example, if you want to know whether an element satisfies a condition, instead of going through each value, the element can be gathered into a list and tested for the specified condition.

```
In[19]:= primelist=Range[100];Map[PrimeQ,primelist]
Out[19]= {False,True,True,False,True,False,True,False,False,False,True,False,
True,False,False,False,True,False,True,False,False,False,False,False,True,False,
False,False,False,True,False,True,False,False,False,False,False,True,False,
False,False,True,False,True,False,False,False,True,False,False,False,False,
False,True,False,False,False,False,False,True,False,True,False,False,False,
False,False,True,False,False,False,True,False,True,False,False,False,False,
False,True,False,False,False,True,False,False,False,False,False,True,False,
False,False,False,False,False,False,False,True,False,False,False}
```

The previous example created a list from 1 to 100 and then tested which of the numbers satisfies the condition of being a prime number with the PrimeQ function. Other functions can be used to test different conditions with numbers and strings. Also, a more specific function for testing logical relations in a list can be used (MemberQ, SubsetQ).

## Defining Own Functions

User functions can be written to perform repetitive tasks and reduce the size of a program. Segmenting the code into functions allows you to create pieces of code that perform a certain task. Functions can receive data from outside when called through parameters and return a fixed result.

A function can be defined with the set or set delayed symbol, but remember, using the set symbol assigns the result to the definition. To define a function, first write the name or symbol, followed by the reference variable and an underscore. As with cases, the underscore tells Mathematica that you are dealing with a dummy variable. As a warning, defined functions cannot have space between letters. Functions can also receive more than one argument.

```
In[20]:= MyF[z_]:=12+2+z;MyF2[x_,z_]:=z/x
```

Now, you can call the function with different z values.

```
In[21]:= List[MyF[1],MyF[324],MyF[5432],MyF2[154,1],MyF2[14,4],MyF2[6,9]]
Out[21]= {15,338,5446, $\frac{1}{154}$ , $\frac{2}{7}$ , $\frac{3}{2}$ }
```

Also, another way to write functions is to write compound functions. This concept is similar to compound expressions; expressions of different classes are written within the definition. Each computation can or cannot be ended with a semicolon. The following example shows the concept.

```
In[22]:= StatsFun[myList_]:= {Max@myList, Min@myList, Mean@myList, Median
@myList, Quantile@@{myList, 1}(*25 percent*)}(*to write a function with
multiple arguments with shorthand notation use curly braces*)}
```

You can also send a list as an argument.

```
In[23]:= myList=Table[m-2,{m,-2,10}];
StatsFun[myList]
Out[24]= {8,-4,2,2,8}
```

You can have multiple operations within a function, with the option to create conditions for the arguments to meet. To write a condition, use the dash and semicolon (/;) symbols. When the condition is true, the function is evaluated; otherwise, if the condition is not true, the function is not evaluated. Compound functions need to be grouped; otherwise, Mathematica treats them as though they are outside the body of the whole function.

The next example creates a function that tells you if an arbitrary string is a palindrome, which is when the word is the same when written backward.

```
In[25]:= PalindromeWord[string_;/StringQ@string==True]:=(*we can check if
the input is really a string*)
(ReverseWord=StringJoin[Reverse[Characters[string]]]);
(*here we separate the characters, reverse the list and join them into a
string*)
ReverseWord==string (*then we test if the word is a palindrome, the output
of the whole function will be True or False*)
```

Let's test the new function.

```
In[26]:= PalindromeWord/@{"hello","room","jhon","kayak","civic","radar"}
Out[26]= {False,False,False,True,True,True}
```

When you have a local assignment on a compound function or functions, the symbols used are still assigned, so if the symbol(s) are called outside the function, it can cause coding errors. One thing to consider is that you can clear the function and local symbols when the function is no longer used. Clearing only the function name does not remove local assignments. Another solution is to declare variables inside a module since the variables are only locally treated, as shown in the following form.

```
In[27]:= MyFunction[a0_,b0_]:=Module[{m=a0,n=b0},(*local variables*)m+n
(*body of the module*)](*end of module*)
In[28]:= Clear[MyF,MyF2,StatsFun,PalindromeWord,ReverseWord] (*To remove
tag names of the functions and local symbols *)
```

## Pure Functions

Pure functions, also known as anonymous functions, are a powerful feature of the Wolfram Language. They allow the execution of a function without referencing a name and can be explicitly assigned to an operation. Arguments within pure functions are denoted with a hashtag (#). To refer to a specific argument, append a number to the hashtag (e.g., #1, #2, (#3, ... for the first, second, third, ... argument). An ampersand (&) is used at the end of the definition to signify the use of the hashtag references. Pure functions can be constructed with the Function keyword or using the shorthand notation of hashtag and ampersand.

```
In[29]:= Function[#^-1][z]==#^-1&[z]
#^-1&[z] (*both expression mean 1/z*)
Out[29]= True
Out[30]= 1/z
```

Some examples of pure functions.

```
In[31]:= {#^-1&[77],#1+#2-#3&[x,y,z]} (*we can imagine that #1,#2,#3 are the
1st,2nd and 3rd variables*),Power[E,#]&[3]}
Out[31]= {1/77,x+y-z,E^3}
```

You can use pure functions along with Map and Apply to pass each argument of a list to a specific function. The # represents each element of the list, and the & represents that # is filled and tested for the elements of the list.

```
In[32]:= N[#]&/@ {1,1,1,12,3,1}
Sqrt[#]&/@{-1,2,4,16}
Out[32]= {1.,1.,1.,12.,3.,1.}
Out[33]= {I,Sqrt[2],2,4}
```

Code can be written more compactly using Apply and pure functions, as shown in the next example. You can select the numbers bigger than 10.

```
In[34]:= Select@@{{1,22,41,7,62,21},#>10&}
Out[34]= {22,41,62,21}
```

## Indexed Tables

You can create and display results in tables to provide a quick way to observe and manage a group of related data, which leads to how to create tables in the Wolfram Language, such as giving titles to columns and names to rows. A series of examples to help you learn the essentials of using the tables so that you can present your data properly are featured in this section.

## Tables with the Wolfram Language

Tables are created with nested lists, and those lists are shown with TableForm.

```
In[35]:= table1={{ "Dog", "Wolf"}, {"Cat", "Leopard"}, {"Pigeon", "Shark"}};
TableForm[table1]
Out[36]//TableForm=
Dog      Wolf
Cat      Leopard
Pigeon   Shark
```

The format of TableForm is ["list", options]. Formatting options let you justify the columns of tables in three ways: left, center, and right. In the next example, the contents of the table are centered.

```
In[37]:= TableForm[table1,TableAlignments->[RightArrow]Right]
Out[37]//TableForm=
Dog      Wolf
Cat      Leopard
Pigeon   Shark
```

Titles can be added with the `TableHeadings` option command and by specifying whether the rows and column labels are exposed or just one of them. Choosing the `Automatic` option gives index labels to the rows and columns. Remember to write strings between the apostrophes or to use `ToString`.

```
In[38]:= TableForm[table1,TableHeadings->{{"Row 1","Row 2","Row
3"},{"Column 1","Column 2"}}]
Out[38]//TableForm=
      | Column 1  Column 2
-----|-----
Row 1 | Dog      Wolf
Row 2 | Cat      Leopard
Row 3 | Pigeon   Shark
```

Labeled rows and columns can be customized with desired names.

```
In[39]:= colname={"Domestic Animals","Wild Animals"};
rowname={"Animal 1","Animal 2","Animal 3"};
TableForm[table1,TableHeadings->{rowname,colname}]
Out[41]//TableForm=
      | Domestic Animals  Wild Animals
-----|-----
Row 1 | Dog                  Wolf
Row 2 | Cat                  Leopard
Row 3 | Pigeon              Shark
```

The same concept applies to labeling just columns or rows by typing `None` on the rows or columns option.



```
In[42]:= TableForm[table1,TableHeadings->{None,{"Domestic Animals","Wild Animals"}}]
```

```
Out[42]//TableForm=
```

```
Domestic Animals    Wild Animals
```

Dog	Wolf
Cat	Leopard
Pigeon	Shark

Automated forms of tables can be created with the use of Table and Range. By applying the Automatic option in the TableHeadings, you can create indexed labels for the data.

```
In[43]:= tabData={Table[i,{i,7}],Table[5^i,{i,7}]];TableForm[tabData,TableHeadings->Automatic]
```

```
Out[43]//TableForm=
```

	1	2	3	4	5	6	7
1	1	2	3	4	5	6	7
2	5	25	125	625	3125	15625	78125

For exhibit reasons, a table can be transposed too.

```
In[44]:= TableForm[Transpose[tabData],TableHeadings->Automatic]
```

```
Out[44]//TableForm=
```

	1	2
1	1	2
2	2	25
3	3	125
4	4	625
5	5	3125
6	6	15625
7	7	78125

Another useful tool is Grid, which displays a list or a nested list in tabular format. Like TableForm, Grid can also be customized to exhibit data more properly.

---

**Note** Grid works with any expression.

---

```
In[45]:= tabData2=Table[{i,Exp[i],N@Exp[i]},{i,7}];
```

```
Grid[tabData2]
```

```
Out[46]=
```

i	Exp <sup>i</sup>	Numeric approx.
1	e	2.71828
2	e <sup>2</sup>	7.38906
3	e <sup>3</sup>	20.0855
4	e <sup>4</sup>	54.5982
5	e <sup>5</sup>	148.413
6	e <sup>6</sup>	403.429
7	e <sup>7</sup>	1096.63

To add headers, insert them in the original list as strings and in position 1.

```
In[47]:= Grid[Insert[tabData2,{"i","Expi","Numeric approx."},1]]
```

```
Out[47]=
```

i	Exp <sup>i</sup>	Numeric approx.
1	e	2.71828
2	e <sup>2</sup>	7.38906
3	e <sup>3</sup>	20.0855
4	e <sup>4</sup>	54.5982
5	e <sup>5</sup>	148.413
6	e <sup>6</sup>	403.429
7	e <sup>7</sup>	1096.63

You can add dividers and spacers too. With Dividers and Spacing, you can divide or space the y and x axes.

```
In[48]:= Grid[Insert[tabData2,{"i","Expi","Numeric approx."},1],
Dividers->{All,False},Spacings->{1,1}]
Out[48]=
```

i	Exp <sup>i</sup>	Numeric approx.
1	e	2.71828
2	e <sup>2</sup>	7.38906
3	e <sup>3</sup>	20.0855
4	e <sup>4</sup>	54.5982
5	e <sup>5</sup>	148.413
6	e <sup>6</sup>	403.429
7	e <sup>7</sup>	1096.63

Background can be added with the Background option. This option allows specific parts of the table or column table to be colored.

```
In[49]:= Grid[Insert[tabData2,{"i","Exp i","Numeric approx."},1],Dividers ->
{All,False},Spacings -> {Automatic,0},Background -> {{LightYellow, None, LightBlue}}]
Out[49]=
```

i	Exp <sup>i</sup>	Numeric approx.
1	e	2.71828
2	e <sup>2</sup>	7.38906
3	e <sup>3</sup>	20.0855
4	e <sup>4</sup>	54.5982
5	e <sup>5</sup>	148.413
6	e <sup>6</sup>	403.429
7	e <sup>7</sup>	1096.63

## Associations

Associations are fundamental in developing the Wolfram Language; associations are used to index lists or other expressions and create more complex data structures. Associations, much like dictionaries in many other programming languages, are a more structured construct that allows you to provide a process for creating pairs of keys and values. Later, you see that they are important for handling datasets in the Wolfram Language.

Associations are of the form `Association["key_1" → val_1, key_2 → val_2 ...]` or `<|"key_1" → "val_1", "key_2" → "val_2" ... |>`; they associate a key to a value. Keys and values can be any expression. The `Association` command is used to construct an association, or you can use the symbolic entry `<| --- |>`.

```
In[50]:= Associt=<|1->"a",2->"b",3->"c"|> (*is the same as Association
[a\[RightArrow]"a",b\[RightArrow]"b",c\[RightArrow]"c"]*)
Associt2=Association[dog->"23", "score"->\[Pi]*\[Pi],2*2->Sin[23 Degree]]
Out[50]= <| 1 → a, 2 → b, 3 → c |>
Out[51]= <| dog → 23, score →  $\pi^2$ , 4 → Sin[23°] |>
```

Entries in an association are ordered, so data can be accessed based on the key of the value or by the position of the entries in the association, like with lists. The position is associated with the values (position of the entries), not the keys, as the order of the keys is not always preserved.

```
In[52]:= Associt[1](*this is key 1 *)
Associt2[[2]] (*this is position of key 2, which is  $\pi^2$  *)
Out[52]= a
Out[53]=  $\pi^2$ 
```

As seen in the latter example, the position is associated with the values, not the key. So, if you want to show parts of the association, use the semicolon.

```
In[54]:= Associt[[1;2]]
Associt2[[2;2]]
Out[54]= <|1→a,2→b|>
Out[55]= <|score→  $\pi^2$  |>
```

Values and keys can be extracted with the Keys and Values commands.

```
In[56]:= Keys@Associt2
Values@Associt2
Out[56]= {dog, score, 4}
Out[57]= {23,  $\pi^2$ , Sin[23 °]}
```

You get an error if you ask for a key without a proper reference.

```
In[58]:= Associt["a"](*there is no "a" key in the association, thus
the error*)
Out[58]= Missing[KeyAbsent,a]
```

Associations can also be associations. The next example shows how to associate associations, thus producing an association of associations. This concept is basic for understanding how a dataset works in the Wolfram Language.

```
In[59]:= Association[Associt,Associt2]
Out[59]= <| 1  $\rightarrow$  a, 2  $\rightarrow$  b, 3  $\rightarrow$  c, dog  $\rightarrow$  23, score  $\rightarrow$   $\pi^2$ , 4  $\rightarrow$  Sin[23°] |>
```

You can also make different associations with lists using AssociationThread. The keys correspond to the first argument and the values to the second. AssociationThread threads a list of keys to a list of values like the next form: <|{"key\_1", "key\_2", "key\_3" ...}  $\rightarrow$  {"val\_1", "val\_2", "val\_3" ...} |>. The latter form can be seen as a list of keys marking a list of values. When you have defined the lists of keys and values, the command can associate a list with another list. You can also create a list of associations to read keys as a row and a column.

```
In[60]:= AssociationThread[{"class", "age", "gender", "survived"}, {"Economy", 29, "female", True}]
Out[60]= <| class  $\rightarrow$  Economy, age  $\rightarrow$  29, gender  $\rightarrow$  female, survived  $\rightarrow$  True |>
```

You can construct the list of keys and values.

```
In[61]:= keys={"class", "age", "gender", "boarded"};
values={"Economy", 29, "female", True};
AssociationThread@@{keys, values}
Out[63]= <| class  $\rightarrow$  Economy, age  $\rightarrow$  29, gender  $\rightarrow$  female,
boarded  $\rightarrow$  True |>
```

More complex structures can be created with associations. For example, the next association creates a data structure based on the information about a sports car, with the model name, engine, power, torque, acceleration, and top speed.

```
In[64]:= Association@{"Model name" -> "Koenigsegg CCX",
"Engine" -> "Twin supercharged V8",
"Power" -> "806 hp",
"Torque" -> "5550 rpm",
"Acceleration 0-100 km/h" -> "3.2 sec",
"Top speed" -> "395 Km/h"}
Out[64]= <|Model name→Koenigsegg CCX, Engine→Twin supercharged V8,
Power→806 hp, Torque→5550 rpm, Acceleration 0-100 km/h→3.2 sec, Top
speed→395 Km/h|>
```

You can see how labels and their elements are created in a grouped way. In addition to that, it is shown how the curly braces mark how each row can arrange the key/value pair.

## Dataset Format

Associations are an essential part of making structured forms of data. Datasets in the Wolfram Language offer a way to organize and exhibit hierarchical data by providing a method for accessing data inside a dataset. This section features examples of how to convert lists, nested lists, and associations to a dataset. It also covers how to add values, access values in a dataset, drop and delete values, map functions over a dataset, deal with duplicate data, and apply functions by row or column.

## Constructing Datasets

Datasets are for constructing hierarchical data frameworks, where lists, associations, and nested lists have an order. Datasets are useful for exhibiting large data in an accessible, structured format. Datasets can show enclosed structures in a sharp format with row headers, column headers, and numbered elements. Having the data as a dataset allows you to look at the data in multiple ways.

Datasets can be constructed in four forms.

- A list of lists; a table with no denomination in rows and columns
- A list of associations, a table with labeled columns; a table with repeated keys and different or same values
- An association of lists, a table with labeled rows; a table with different keys and different or same values
- Association of associations; a table with labeled rows and columns

The most common form to create a new dataset is from a list of lists. Create a list within the curly braces {} using the Dataset function. Each brace represents the parts of the table. Figure 3-1 shows the output of the Dataset function.

```
In[65]:= Dataset@{{"Jhon",23,"male","Portugal"}, {"Mary",30,"female","USA"}, {"Peter",33,"male","France"}, {"Julia",53,"female","Netherlands"}, {"Andrea",45,"female","Brazil"}, {"Jeff",24,"male","Mexico"}}
Out[65]=
```

Jhon	23	male	Portugal
Mary	30	female	USA
Peter	33	male	France
Julia	53	female	Netherlands
Andrea	45	female	Brazil
Jeff	24	male	Mexico

**Figure 3-1.** Dataset object created from the input code

By hovering the mouse cursor over the elements of the dataset, you can see their position in the lower-left corner. The name France corresponds to row 3 and column 4. The notation of a dataset is first rows, then columns. If you have labeled columns, rows, or both, you see the column name and row name instead of the numbers.

Constructing a dataset with a list of associations is performed by creating associations first with repeated keys and then enclosing them in a list. First, create the associations; the repeated keys specify each column header. The values represent the contents of the columns. Datasets have a head expression of Dataset.

```

In[66]:=
Dataset@{
<|"Name"->"Jhon", "Age"->23, "Gender"->"male", "Country"->"Portugal"|>,
<|"Name"->"Mary", "Age"->30, "Gender"->"female", "Country"->"USA"|>,
<|"Name"->"Peter", "Age"->33, "Gender"->"male", "Country"->"France"|>,
<|"Name"->"Julia", "Age"->53, "Gender"->"female", "Country"->"Netherlands"|>,
<|"Name"->"Andrea", "Age"->45, "Gender"->"female", "Country"->"Brazil" |>,
<|"Name" -> "Jeff", "Age" -> 24, "Gender" -> "male", "Country" -> "Mexico"
|>>(*Head @ % *)
Out[66]=

```

As seen in Figure 3-2, Mathematica recognizes that Name, Age, Gender, and Country are column headers, which is why the color of the box is different.

Name	Age	Gender	Country
Jhon	23	male	Portugal
Mary	30	female	USA
Peter	33	male	France
Julia	53	female	Netherlands
Andrea	45	female	Brazil
Jeff	24	male	Mexico

**Figure 3-2.** Dataset with column headers


When passing the cursor over the column labels, they are highlighted in blue, thus making it possible to click the name of the label, and then it produces only the selected label and not the whole dataset, as seen in Figure 3-3.





Jhon
Mary
Peter
Julia
Andrea
Jeff

**Figure 3-3.** Column name selected in the dataset

When this happens, the name of the column also appears. To return to the whole dataset, hit the spreadsheet icon  in the upper-left corner or the name All. This type of layout is practical when dealing with a big set of rows and columns, and you want to focus only on a few sections of the dataset.

In an association of lists, the keys represent the label of the rows, and the values are the list of the elements of the rows; then, you associate the whole block. The next block of code generates an association of a list.

---

**Note** The same is true here. Whenever you click a row's name, it only displays that row.

---

```
In[67]:= Dataset@
<|"Subject A"->{"Jhon",23,"male","Portugal"},
"Subject B"->{"Mary",30,"female","USA"},
"Subject C"->{"Peter",33,"male","France"},
"Subject D"->{"Julia",53,"female","Netherlands"},
"Subject E"->{"Andrea",45,"female","Brazil"},
"Subject F"->{"Jeff",24,"male","Mexico"}|>
Out[67]=
```

As seen in Figure 3-4, the rows are now labeled.

Subject A	Jhon	23	male	Portugal
Subject B	Mary	30	female	USA
Subject C	Peter	33	male	France
Subject D	Julia	53	female	Netherlands
Subject E	Andrea	45	female	Brazil
Subject F	Jeff	24	male	Mexico

**Figure 3-4.** Dataset with labeled rows

Row labels are recognized and displayed in the color box. When selecting the row's label, it display only that row, as shown in Figure 3-5.

 Subject E

Andrea	45	female	Brazil
--------	----	--------	--------

**Figure 3-5.** Subject E row selected


In an association of associations, the repeated keys of the association of associations are the column labels and the values of the dataset. In the second association, the keys are the labels of the rows, and the first associations are the values of the second association. The next example clarifies this.

```
In[68]:= Dataset@
<|"Subject A"-><|"Name"->"Jhon", "Age"->23, "Gender"->"male", "Country"->
"Portugal"|>, "Subject B"-><|"Name"->"Mary", "Age"->30, "Gender"->
"female", "Country"->"USA"|>, "Subject C"-><|"Name"->"Peter", "Age"->33,
"Gender"->"male", "Country"->"France"|>,
"Subject D"-><|"Name"->"Julia", "Age"->53, "Gender"->"female", "Country"->
"Netherlands"|>, "Subject E"-><|"Name"->"Andrea", "Age"->45, "Gender"->
"female", "Country"->"Brazil"|>, "Subject F"-><|"Name"->"Jeff", "Age"->
24, "Gender"->"male", "Country"->"Mexico"|>|>
Out[68]=
```

	Name	Age	Gender	Country
Subject A	Jhon	23	male	Portugal
Subject B	Mary	30	female	USA
Subject C	Peter	33	male	France
Subject D	Julia	53	female	Netherlands
Subject E	Andrea	45	female	Brazil
Subject F	Jeff	24	male	Mexico

**Figure 3-6.** Dataset with names in rows and columns

As can be seen in Figure 3-6, the rows and columns are now labeled. Like the previous examples, the column and row labels are recognized and displayed in the color box. When selecting the label of the row or a column, it displays only that row or column, as seen in Figure 3-7.

 Subject F

Name	Age	Gender	Country
Jeff	24	male	Mexico

**Figure 3-7.** Only a row selected

If you select only a particular value, then that value is solely displayed. Figure 3-8 shows its form.

 Subject F Name

Jeff
------

**Figure 3-8.** Name for subject F

Creating a dataset from associations of associations is best for compact datasets because sometimes it can get messy to extract values and keys. However, the best approach is the one that works best for you.

## Accessing Data in a Dataset

Mathematica gives each element a unique index; so if you are interested in selecting data from a dataset, assign a symbol to the dataset and proceed to specify each output in the next form. The first and second positions of the arguments represent row and column [nth row, mth column]. So, to extract data based on a column name or a set of columns, enclose the columns in brackets. You can also use double-bracket notation. If only one argument is received, it is only the rows. First, let's create the dataset.

```
In[69]:=Dst=Dataset@{
<|"Name"->"Jhon", "Age"->23, "Gender"->"male", "Country"->"Portugal"|>,
<|"Name"->"Mary", "Age"->30, "Gender"->"female", "Country"->"USA"|>,
<|"Name"->"Peter", "Age"->33, "Gender"->"male", "Country"->"France"|>,
<|"Name"->"Julia", "Age"->53, "Gender"->"female", "Country"->"Netherlands"|>,
<|"Name"->"Andrea", "Age"->45, "Gender"->"female", "Country"->"Brazil"|>,
<|"Name"->"Jeff", "Age"->24, "Gender"->"male", "Country"->"Mexico"|>};
```

The notation `[[ ]]` works the same as the special character for double brackets (`[[]]`). Also, you can select data using the specific keys of the value, as shown in Figure 3-9.

```
In[70]:= Dst[[1,2]](*This is for row 1,column 2*)
Dst[1](*row 1*)
Out[70]= 23
Out[71]=
```

Name	Jhon
Age	23
Gender	male
Country	Portugal

**Figure 3-9.** Row 1 for Dst

Let's look at the following and Figure 3-10.

```
In[72]:= Dst[1;;3](*to manipulate data of the column try Dst[1;;3,1;;3]*)
Out[72]=
```

Name	Age	Gender	Country
Jhon	23	male	Portugal
Mary	30	female	USA
Peter	33	male	France

**Figure 3-10.** Values from rows 1 to 3 and columns 1 to 3

This case selected data from positions 1 to 3, from John to Peter. The same is applied to columns.

You can also show specific columns and maintain all the fixed rows with their keys. The same process is applied when having a label in each row. Typing All means all the elements in the column or the row. The output is shown in Figure 3-11.

```
In[73]:= Dst[All,{"Name","Age"}] (*If more than 1 column label is added
then enclosed the labels by curly braces.*)
Out[73]=
```

Name	Age
Jhon	23
Mary	30
Peter	33
Julia	53
Andrea	45
Jeff	24

**Figure 3-11.** Values for column name and age

Alternatively, you can extract a column or a row as a list to better manipulate them in the Wolfram Language. To do that you need to use the `Normal` function and the `Values` command. Remember that you are dealing with associations, so if you want the values, you use the `Values` command and then `Normal` to convert it to a normal expression.

```
In[74]:= Normal@Values@Dst[All,{"Name","Age"}]( *values of the name and age
columns*)
```

```
Out[74]= {{Jhon,23},{Mary,30},{Peter,33},{Julia,53},{Andrea,45},{Jeff,24}}
```

It is the same idea for the rows: if they have a label, you can use them.

```
In[75]:= Normal@Values@Dst[[1,All]]
```

```
Out[75]= {Jhon,23,male,Portugal}
```

The result is the same if you first do `Normal` and then `Values`.

```
In[76]:= Values@Normal@Dst[[1,All]]
```

```
Out[76]= {Jhon,23,male,Portugal}
```

Another function that can be used is `Query`, a specialized function that works with datasets. Queries must be applied to the symbol of the dataset or directly to the dataset. Queries are helpful because they allow easy selectivity of the values; you can extract rows or columns and get individual records.

```
In[77]:= Query[All,"Country"]@Dst
```

```
Query[3]@%
```

```
Out[77]=
```

Figure 3-12 shows that you can extract columns and values with `Query`.

Portugal
USA
France
Netherlands
Brazil
Mexico

**Figure 3-12.** *Country values*

```
Out[78]= France
```

Another function that works more intuitively is `Take`, in which you can specify the symbol of the dataset and then how many rows and columns to display. `Take` comes in handy when dealing with large datasets, and you want to only view a specific part of the data.

```
In[79]:= Take[Dst,2] (*First 2 rows*)
(*Take[Dst,3,3] First 3 rows and columns*)
Out[79]=
```

Figure 3-13 shows you can use `Take` as an alternative.

Name	Age	Gender	Country
Jhon	23	male	Portugal
Mary	30	female	USA

**Figure 3-13.** *First two rows of a dataset*

## Adding Values

Now that you have examined how to access the elements of a dataset, you can add new values to the dataset. You can add rows with `Append` or `Prepend`, but remember that `AppendTo` and `PrependTo` can be used too. However, they assign the new result to the assigned variable. `Append` adds at the last and `Prepend` at the first.

To add a row, you would need to write the new row like you write the associations with repeated keys, calling the dataset and then the function, followed by the new row, as shown in Figure 3-14.

```
In[80]:= Dst[Append[<|"Name"->"Anya", "Age"->19, "Gender"->
"female", "Country"->"Russia"|>]]
Out[80]=
```

Name	Age	Gender	Country
Jhon	23	male	Portugal
Mary	30	female	USA
Peter	33	male	France
Julia	53	female	Netherlands
Andrea	45	female	Brazil
Jeff	24	male	Mexico
Anya	19	female	Russia

**Figure 3-14.** New row added at the end of the dataset

The operator form of the `Append` function was used in this case. Operator forms in the Wolfram Language allows for a more concise and readable code syntax. They essentially allow function to be used directly without square brackets. This form can be used with other function, like `Apply`, to make expression with a more natural representation. For example, to add a new row at the top of the dataset, try using the code, `Dst@Prepend[<|"Name"->"Anya", "Age"->19, "Gender"->"female", "Country"->"Russia"|>]`, which is the same as `Dst[Prepend[<|"Name"->"Anya", "Age"->19, "Gender"->"female", "Country"->"Russia"|> ]]`.



Adding a new column of only single values can be done by simply assigning a value to the side of the columns of the dataset with the key name, which is the column name. Figure 3-15 shows the new column added.

```
In[81]:= Dst[All,Prepend["ID number"->1]]
Out[81]=
```

ID number	Name	Age	Gender	Country
1	Jhon	23	male	Portugal
1	Mary	30	female	USA
1	Peter	33	male	France
1	Julia	53	female	Netherlands
1	Andrea	45	female	Brazil
1	Jeff	24	male	Mexico

**Figure 3-15.** ID column added

To add a list of values as a column, first create a list of values. Next, use AssociationThread to associate each value with the same key, creating an association of values for the repeated key. Then you create a dataset of the new association and combine it with the original dataset with the Join function. This merges expressions of the same head.

```
In[82]:= Id={1,2,3,4,5,6};(*our list of values*)
ID=AssociationThread["ID"->#]&/@Id (*the process is threaded in the list*)
Out[82]= {<|ID->1|>,<|ID->2|>,<|ID->3|>,<|ID->4|>,<|ID->5|>,<|ID->6|>}
```

Each element needs to be associated one by one for the later block because AssociationThread suppresses repeated keys, so you would only have one association, and you need to have a repeated key marking different values.

Next, create the new dataset with the same key shown in Figure 3-16.

```
In[83]:= Dataset[ID]
Out[83]=
```

ID
1
2
3
4
5
6

**Figure 3-16.** ID column dataset

Finally, join the same objects; here, Join is used with a level of specification of 2 because the new dataset is a sublist of depth 2. If you want to add the column on the left side, the new column goes first, followed by the dataset; for the right side, it is the opposite. Figure 3-17 shows the output dataset.

```
In[84]:= Join[%,Dst,2]
Out[84]=
```

ID	Name	Age	Gender	Country
1	Jhon	23	male	Portugal
2	Mary	30	female	USA
3	Peter	33	male	France
4	Julia	53	female	Netherlands
5	Andrea	45	female	Brazil
6	Jeff	24	male	Mexico

**Figure 3-17.** ID column added

The previous cases worked with a dataset from a list of associations; since you are working with tagged rows only or tagged rows and columns, adding a row or column is preserved by adding the same structure to the dataset. So, adding a new row to an association of lists would take the form `<| "key" → {elem, ... } |>`; for columns, this would be the process of creating a dataset and joining them. In the case of a list of lists, adding a row would be the same approach but without a key. For the case of association of associations, to add a row would be `<| "key" → <| "key 1" → "val 1", ... | > |>`, and for columns, it would be the same as before, a key associated with a value. Nevertheless, there is no restriction on how data can be accommodated.

Finally, to change unique values, select the item and give it the new content. In the case that you have labels on rows and columns, the original form is still preserved: `"rows", "columns"`. So, if you want to replace Jhon's age, use the `ReplacePart` function by calling the symbol of the dataset and specifying the column tag and then with the new value, which is 50. If you were working with only a row label or a column label, the process would be the same, but using the row or column label and then the number position of the element. Figure 3-18 shows the new value is 50.

```
In[85]:= ReplacePart[Dst,{1,"Age"}->50](*Also using the index will produce  
the same output,that would be {1,2} -> 50*)  
Out[85]=
```

Name	Age	Gender	Country
Jhon	50	male	Portugal
Mary	30	female	USA
Peter	33	male	France
Julia	53	female	Netherlands
Andrea	45	female	Brazil
Jeff	24	male	Mexico

**Figure 3-18.** *Jhon age value changed to 50*

## Dropping Values

You can eliminate the contents of a row or column without deleting the entire table structure. To accomplish this, use the Drop function or the Delete function. When using Drop, you enclose the number of the row or column with { } to delete a unique row or column (see Figure 3-19).

```
In[86]:= Drop[Dst,{1}]( *in the instance we want to delete more than one
then we write m through n dropped {m,n}*)
```

```
Out[86]=
```

Name	Age	Gender	Country
Mary	30	female	USA
Peter	33	male	France
Julia	53	female	Netherlands
Andrea	45	female	Brazil
Jeff	24	male	Mexico

**Figure 3-19.** Drop row 1

Figure 3-19 shows that the first row has been dropped. You can also drop rows and columns at the same time. Figure 3-20 shows the second row and last column dropped.

```
In[87]:= Drop[Dst,{2},{4}]
```

```
Out[87]=
```

Name	Age	Gender
Jhon	23	male
Peter	33	male
Julia	53	female
Andrea	45	female
Jeff	24	male

**Figure 3-20.** New dataset after dropping row 2 and column 4

Another way is to use Delete on a row or column label, as shown in Figure 3-21.

```
In[88]:= Dst[All,Delete["Age"]] (*to delete a row use["label of row",All]*)
Out[88]=
```

Name	Gender	Country
Jhon	male	Portugal
Mary	female	USA
Peter	male	France
Julia	female	Netherlands
Andrea	female	Brazil
Jeff	male	Mexico

**Figure 3-21.** Age column deleted

## Filtering Values

Having the data as a dataset allows you to look at the data in multiple ways. Let's now work with the tagged dataset to better expose how filtering values work. For starters, you use the labeled dataset shown in Figure 3-22.

```
In[89]:= Clear[Dst];(*Let's clear the symbol "Dst" of previous
assignments*)
Dst=Dataset@
<|"Subject A"-><|"Name"->"Jhon", "Age"->23, "Gender"->"male", "Country"->
"Portugal"|>, "Subject B"-><|"Name"->"Mary", "Age"->30, "Gender"->
"female", "Country"->"USA"|>, "Subject C"-><|"Name"->"Peter", "Age"->33,
"Gender"->"male", "Country"->"France"|>,
"Subject D"-><|"Name"->"Julia", "Age"->53, "Gender"->"female", "Country"->
"Netherlands"|>, "Subject E"-><|"Name"->"Andrea", "Age"->45, "Gender"->
"female", "Country"->"Brazil"|>, "Subject F"-><|"Name"->"Jeff", "Age"->24,
"Gender"->"male", "Country"->"Mexico"|>
|>
Out[90]=
```

	Name	Age	Gender	Country
Subject A	Jhon	23	male	Portugal
Subject B	Mary	30	female	USA
Subject C	Peter	33	male	France
Subject D	Julia	53	female	Netherlands
Subject E	Andrea	45	female	Brazil
Subject F	Jeff	24	male	Mexico

**Figure 3-22.** Tagged dataset

As with lists, you can create one or more filter conditions; for example, you can select an age greater than 30 and get a dataset object (see Figure 3-23).

```
In[91]:= Cases[Dst[All,"Age"],x_/;x>30](*also we can select data that
matches exactly 30 with the==sign*)
Out[91]=
```

33
53
45

**Figure 3-23.** *Filtered data from the age column*

Figure 3-23 shows the filtered data. Data can be selected based on True or False results. For that, you can use the Select function. Figure 3-24 shows the selected subjects.

```
In[92]:= Select[Dst[All,"Age"],EvenQ]
Out[92]=
```

Subject B	30
Subject F	24

**Figure 3-24.** *Selected subjects*

The use of pure functions can be applied too. Remember that the #Age resembles the elements in the Age column, as shown in Figure 3-25.

```
In[93]:= Dst[Select[#Age>30&]]
Out[93]=
```

	Name	Age	Gender	Country
Subject C	Peter	33	male	France
Subject D	Julia	53	female	Netherlands
Subject E	Andrea	45	female	Brazil

**Figure 3-25.** *Selected values using pure function syntax*

Also, you can count categorical data values, as shown in Figure 3-26. This is helpful when you want to identify how many types of a class you have in the data. For example, you can count how many females and males are in the dataset.

```
In[94]:= Counts[Dst[All,"Gender"]] (*alternative
form:Dst[Counts,"Gender"]*)
Out[94]=
```

male	3
female	3

**Figure 3-26.** *Count data for class male and female*

More complex groups can be made based on a class; for instance, you can group the dataset by gender, as shown in Figure 3-27.

```
In[95]:= Dst[GroupBy["Gender"],Counts,"Age"]
Out[95]=
```



male	23	1
	33	1
	24	1
female	30	1
	53	1
	45	1

**Figure 3-27.** Data arranged by class and age

As a good practice, clear symbols when they are no longer used.

```
In[96]:= Clear[Dst]
```

# Applying Functions

Functions can be applied to the dataset to get statistics, determine dimensions, or transform the data. Functions can be applied to single columns or a unique element in the data structure. First, let’s create a dataset comprising 10 items, whose columns are the factorial of 1 to 10, a random real number from 1 to 0, and the natural logarithm from 1 to 10. Figure 3-28 shows the new dataset.

```
In[97]:= DataNubr=Dataset@Table[<|"Factorial"->Factorial[i],"Random
number"->RandomReal[{0,1}], "Natural Logarithm"->Log[E,i]|>,{i,1,10}]
Out[97]=
```

Factorial	Random number	Natural Logarithm
1	0.556204	0
2	0.747067	0.693147
6	0.986582	1.09861
24	0.905028	1.38629
120	0.395201	1.60944
720	0.507363	1.79176
5040	0.5893	1.94591
40 320	0.168404	2.07944
362 880	0.904704	2.19722
3 628 800	0.211938	2.30259

**Figure 3-28.** *Numeric dataset*

And now you can compute basic operations on the data, like getting the mean of the factorials and random numbers, as shown in Figure 3-29.

```
In[98]:= DataNumbr[Mean,{"Factorial","Random number"}]/N
Out[98]=
```

Factorial	403 791.
Random number	0.597179

**Figure 3-29.** *Mean for values in Factorial and Random number columns*

Parenthesis and the composition of functions can also be used to relate operations applied to the data by using the @\*(composition) symbol. Figure 3-30 shows the data for random numbers sorted from less to greater.

```
In[99]:= DataNumbr[All,"Random number"]@(Sort@*N)
Out[99]=
```

0.168404	0.211938	0.395201	0.507363	0.556204
0.5893	0.747067	0.904704	0.905028	0.986582

**Figure 3-30.** Sorted data in canonical order

You can apply different functions to the data. As shown in Figure 3-31, the dataset shows numbers in decimal form; otherwise, it would not fit in the square box.

```
In[100]:= DataNumbr[{Total,Max,Min},"Natural Logarithm"]
Out[100]=
```

15.1044	2.30259	0
---------	---------	---

**Figure 3-31.** Total, Max, and Min value for Natural Logarithm column

You can also apply your own functions; let’s use a previously constructed function. Figure 3-32 shows the function you created previously applied to a dataset column.

```
In[101]:= DataNumbr[{StatsFun},"Natural Logarithm"]
Out[101]=
```

2.30259	0	1.51044	1.7006	2.30259
---------	---	---------	--------	---------

**Figure 3-32.** StatsFun applied to the Natural Logarithm column

Functions to restructure the dataset can be applied too, like Reverse, as shown in Figure 3-33.

```
In[102]:= DataNumbr[Reverse,All]
Out[102]=
```

Factorial	Random number	Natural Logarithm
3 628 800	0.211938	2.30259
362 880	0.904704	2.19722
40 320	0.168404	2.07944
5040	0.5893	1.94591
720	0.507363	1.79176
120	0.395201	1.60944
24	0.905028	1.38629
6	0.986582	1.09861
2	0.747067	0.693147
1	0.556204	0

**Figure 3-33.** *Reversed elements of the dataset*

Map can also apply functions, as you saw with lists in the previous sections. The next example maps a function directly into the dataset, as shown in [Figure 3-34](#).

```
In[103]:= Map[Sqrt,DataNumbr]
Out[103]=
```

Factorial	Random number	Natural Logarithm
1	0.745791	0
1.41421	0.86433	0.832555
2.44949	0.993268	1.04815
4.89898	0.95133	1.17741
10.9545	0.62865	1.26864
26.8328	0.712294	1.33857
70.993	0.767659	1.39496
200.798	0.410371	1.44203
602.395	0.951159	1.4823
1904.94	0.460367	1.51743

**Figure 3-34.** The square root function mapped in the dataset

Transposition is an operation that consists of converting columns to rows and rows to columns and can sometimes help you observe data differently. To obtain the transposition of the dataset, use the Transpose function applied to the dataset. Figure 3-35 shows all columns are now rows and displayed compactly because it is a large row.

```
In[104]:= DataNumbr//Transpose
Out[104]=
```

Factorial	{ ... <sub>10</sub> }
Random number	{ ... <sub>10</sub> }
Natural Logarithm	{ ... <sub>10</sub> }

**Figure 3-35.** Dataset values by Mathematica due to large contents

If you click a row, you should get the values for the corresponding row.

## Functions by Column or Row

Another approach is to directly apply a function to the values of a column, and you can specify a rule of transformation. For example, you can round to the smallest integer greater than or equal to all the values in the Natural Logarithm column. Figure 3-36 shows the output.

```
In[105]:= DataNumbr[All,{"Natural Logarithm"->Ceiling}](*The same can be
done using the index number of the columns,DataNumbr*)
Out[105]=
```

Factorial	Random number	Natural Logarithm
1	0.54158	0
2	0.223704	1
6	0.473125	2
24	0.726243	2
120	0.371648	2
720	0.37111	2
5040	0.581207	2
40 320	0.316827	3
362 880	0.254744	3
3 628 800	0.463658	3

**Figure 3-36.** *Ceiling function applied as a rule*

You can apply the square root to the first row. Map can also be used to apply functions to rows. Figure 3-37 shows the output generated

```
In[106]:= DataNumbr[1,Sqrt] (*Map[Sqrt,DataNumbr[1;;2,All]] can also do the
work for the first 2 rows*)
Out[106]=
```

Factorial	1
Random number	0.735921
Natural Logarithm	0

**Figure 3-37.** Output generated from the earlier code

When you want to apply a function to a defined level, you can use MapAt. MapAt has the form MapAt[f, “expr”, {i, j, ...}], where {i, j} means the level of the position, as shown in Figure 3-38.

```
In[107]:= MapAt[Exp,DataNumbr,{1]}(*for first position of row 1 only*)
(*Double semi-colon can be used to define from row to row,try using 4;;6.
Caution you might get big numbers*)
Out[107]=
```

Factorial	Random number	Natural Logarithm
2.71828	1.71872	1
2	0.223704	0.693147
6	0.473125	1.09861
24	0.726243	1.38629
120	0.371648	1.60944
720	0.37111	1.79176
5040	0.581207	1.94591
40 320	0.316827	2.07944
362 880	0.254744	2.19722
3 628 800	0.463658	2.30259

**Figure 3-38.** Exponentiation for the first row only with MapAt

Occasionally, you might encounter duplicate data, making it hard to understand the data, especially if something goes wrong. One approach can be to remove an entire row or column, as you saw in previous sections; but as an alternative, you can use built-in functions that can do the job. The `DeleteDuplicates` function is the most common. `DeleteCases` can be used, too, but it removes data that matches a pattern, in contrast to `DeleteDuplicates`. Let's create a dataset for the example.

```
In[108]:= Sales = Dataset@{
<|"Id" -> 1, "Product" -> "PC", "Price" -> "800 €", "Sale Month" ->
"January"|>,
<|"Id" -> 2, "Product" -> "Smart phone", "Price" -> "255 €", "Sale Month"
-> "January"|>,
<|"Id" -> 3, "Product" -> "Anti-Virus", "Price" -> "100 €", "Sale Month"
-> "March"|>,
<|"Id" -> 4, "Product" -> "Earphones", "Price" -> "78 €", "Sale Month" ->
"February"|>,
<|"Id" -> 5, "Product" -> "PC", "Price" -> "809 €", "Sale Month" ->
"March"|>,
<|"Id" -> 5, "Product" -> "PC", "Price" -> "809 €", "Sale Month" ->
"March"|>,
<|"Id" -> 6, "Product" -> "Radio", "Price" -> "60 €", "Sale Month" ->
"January"|>,
<|"Id" -> 7, "Product" -> "PC", "Price" -> "700 €", "Sale Month" ->
"February"|>,
<|"Id" -> 8, "Product" -> "Mouse", "Price" -> "100 €", "Sale Month" ->
"March"|>,
<|"Id" -> 9, "Product" -> "Keyboard", "Price" -> "125 €", "Sale Month" ->
"January"|>,
<|"Id" -> 10, "Product" -> "USB 64gb", "Price" -> "90 €", "Sale Month" ->
"March"|>,
<|"Id" -> 11, "Product" -> "LED Screen", "Price" -> "900 €", "Sale Month"
-> "February"|>,
<|"Id" -> 11, "Product" -> "LED Screen", "Price" -> "900 €", "Sale Month"
-> "February"|>}
Out[108]=
```



Figure 3-39 reveals two duplicated rows in the dataset: ID numbers 5 and 11. The DuplicateFreeQ function can detect whether the dataset appears to have duplicates. The function returns False when there is duplicate data and True when there is not. It can be applied straight to the dataset, or you can detect the rows that appear to be duplicated.

Id	Product	Price	Sale Month
1	PC	800 €	January
2	Smart phone	255 €	January
3	Anti-Virus	100 €	March
4	Earphones	78 €	February
5	PC	809 €	March
5	PC	809 €	March
6	Radio	60 €	January
7	PC	700 €	February
8	Mouse	100 €	March
9	Keyboard	125 €	January
10	USB 64gb	90 €	March
11	LED Screen	900 €	February
11	LED Screen	900 €	February

**Figure 3-39.** Dataset example for duplicate data

```
Let's check if there are duplicates in rows 1 through 7.  
  
In[109]:= DuplicateFreeQ[Sales[1;;7,All]]  
Out[109]= False
```

Duplicate data was programmatically found in the dataset. You can also check for duplicates by column.

```
In[110]:= Sales[All,{"Id"}]@DuplicateFreeQ
Out[110]= False
```

To delete duplicates, the `DeleteDuplicates` function is used. It can be applied to the dataset, column, or row as a function. The output generated is shown in Figure 3-40.

```
In[111]:= DeleteDuplicates[Sales] (*Datas[All,{"ID"}]@DuplicateFreeQ*)
Out[111]=
```

Id	Product	Price	Sale Month
1	PC	800 €	January
2	Smart phone	255 €	January
3	Anti-Virus	100 €	March
4	Earphones	78 €	February
5	PC	809 €	March
6	Radio	60 €	January
7	PC	700 €	February
8	Mouse	100 €	March
9	Keyboard	125 €	January
10	USB 64gb	90 €	March
11	LED Screen	900 €	February

**Figure 3-40.** Dataset without duplicates

An alternative is to use GroupBy to identify which data is duplicated in the dataset. Notice in Figure 3-41 that the repeated data is stacked together.

```
In[112]:= GroupBy[Sales,"Id"]
Out[112]=
```

	Id	Product	Price	Sale Month
1	1	PC	800 €	January
2	2	Smart phone	255 €	January
3	3	Anti-Virus	100 €	March
4	4	Earphones	78 €	February
5	5	PC	809 €	March
	2 total ›			
6	6	Radio	60 €	January
7	7	PC	700 €	February
8	8	Mouse	100 €	March
9	9	Keyboard	125 €	January
10	10	USB 64gb	90 €	March
11	11	LED Screen	900 €	February
	2 total ›			

*Figure 3-41. Dataset grouped by duplicates*

## Joining and Merging Datasets

Combining multiple datasets into one based on shared attributes is a frequent task. This process can be achieved depending on how a dataset should be joined. The three different functions that operate on datasets are Join, JoinAcross, and Merge.

The first function combines two datasets end-to-end, effectively concatenating them into a single dataset (see Figure 3-42).

```
In[113]:= dataset1={<|"a"->1,"b"->2|>,<|"a"->3,"b"->4|>};
dataset2={<|"a"->5,"b"->6|>};
Join[dataset1,dataset2]//Dataset
Out[116]=
```

a	b
1	2
3	4
5	6

**Figure 3-42.** Dataset grouped by the Join function

The second function combines datasets on a specified key or keys, similar to how relational databases join tables based on common keys (see Figure 3-43). Similar to operations from relational databases like join, left join, right join, inner join, outer join, and more.

```
In[117]:= dataset3={<|"ID"->1,"Value"->"A"|>,<|"ID"->2,"Value"->"B"|>};
dataset4={<|"ID"->1,"Score"->95|>,<|"ID"->2,"Score"->90|>};
JoinAcross[dataset3,dataset4,"ID"]//Dataset
Out[119]=
```

ID	Value	Score
1	A	95
2	B	90

**Figure 3-43.** Dataset combined by the JoinAcross function

The third function combines datasets, using a function *f* to combine values with the same key, returning a single value (see Figure 3-44).

```
In[120]:= Merge[Dataset[JoinAcross[dataset3,dataset4,"ID"]],Total]
Out[120]=
```

ID	3
Value	"A" + "B"
Score	185

**Figure 3-44.** Dataset combined by the Merge and Total functions of each key

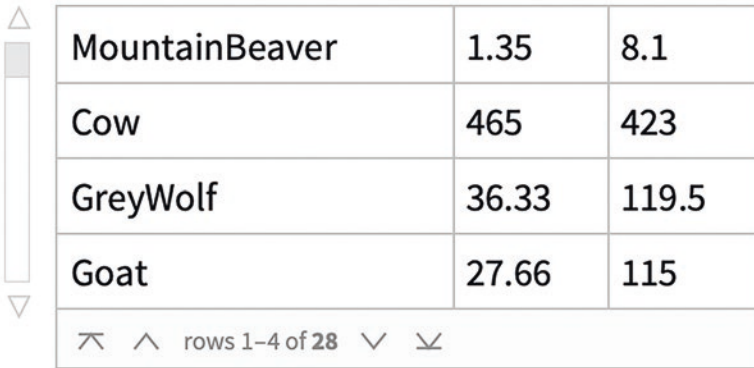
## Customizing a Dataset

Datasets can be customized depending on how you want to show the data. Working with datasets can be personalized based on preferences. To explore this, the next block loads example data from the Wolfram reference servers to discover how to personalize data for your needs. When loading data from the server, depending on your Internet connection, it might pop up a loading frame trying to access the Wolfram servers.

Let's load the data by using `ExampleData` and then choosing statistics of animal weights and converting the list into a dataset. By using the `MaxItem` option, you can display how many rows or columns to exhibit from the dataset. The first four rows and the first three columns are shown in this example. When viewing the dataset, scroll

bars appear on the left and top sides; use them to move over the dataset. Alternatively, you can align the contents on the left, center, or right sides. In Figure 3-45, only the left scrollbars appear.

```
In[121]:= AnimalData=ExampleData[{"Statistics","AnimalWeights"}];
Dataset[AnimalData,MaxItems->{4,3},Alignment->Center] (*To align a
sole column,Alignment-> "Col_name" -> Left*)
Out[121]=
```




MountainBeaver	1.35	8.1
Cow	465	423
GreyWolf	36.33	119.5
Goat	27.66	115

rows 1-4 of 28

**Figure 3-45.** *Animal dataset*

The Background option is used to color the dataset's contents; the colors of the notation {row, col} are preserved. To paint the whole data, enter only the color. To paint by row or column, enter the colors as a nested list—that is, {{“color\_row1”, “color\_row2”, ... }, {“color\_col1”, “color\_col2”, ... } }. Mixing colors can also be done by nesting the nested colors. For specific values, the position of the values would need to be entered. The next example colors the first two columns, as shown in Figure 3-46.

```
In[122]:= Dataset[AnimalData,MaxItems->{4,3},Background-> {{None},{LightBlue,
LightYellow}},ItemSize->{12}]
Out[122]=
```




MountainBeaver	1.35	8.1
Cow	465	423
GreyWolf	36.33	119.5
Goat	27.66	115

rows 1-4 of 28

**Figure 3-46.** Columns 1 and 2 colored

For particular values, the position of the values would need to be entered. Another option is the size of the items, which is controlled with the `ItemSize` option. If you want to edit the same options but with headers, you would use `HeaderAlignment` for placing the text left, center, or right; `HeaderSize` for the size of the titles; and `ItemStyle` for the style of the font of the items. Figure 3-47 shows the dataset in bold style.

```
In[123]:= Dataset[AnimalData,MaxItems->{4,3},Background->{{4,3}->
Yellow},ItemSize->{12},ItemStyle->Bold]
Out[123]=
```



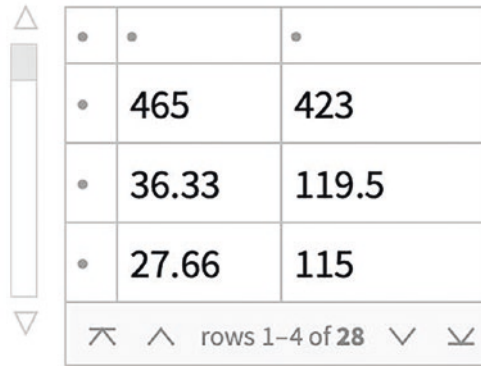
<b>MountainBeaver</b>	<b>1.35</b>	<b>8.1</b>
<b>Cow</b>	<b>465</b>	<b>423</b>
<b>GreyWolf</b>	<b>36.33</b>	<b>119.5</b>
<b>Goat</b>	<b>27.66</b>	<b>115</b>

rows 1-4 of 28

**Figure 3-47.** Dataset with bold style

Another useful option is `HiddenItems`, which hides items that should not be displayed. Therefore, to hide row 1 and column 1, use `HiddenItems`  $\rightarrow$  `{"row #", "col #"}`. Columns can be hidden with their associated labels. Figure 3-48 illustrates the form of suppressed rows and columns in the dataset. For specific values, nest the value's position and try `HiddenItems`  $\rightarrow$  `{{2,3}}`.

```
In[124]:= Dataset[AnimalData,MaxItems->{4,3},HiddenItems->{1,1}]
Out[124]=
```



•	•	•
•	465	423
•	36.33	119.5
•	27.66	115

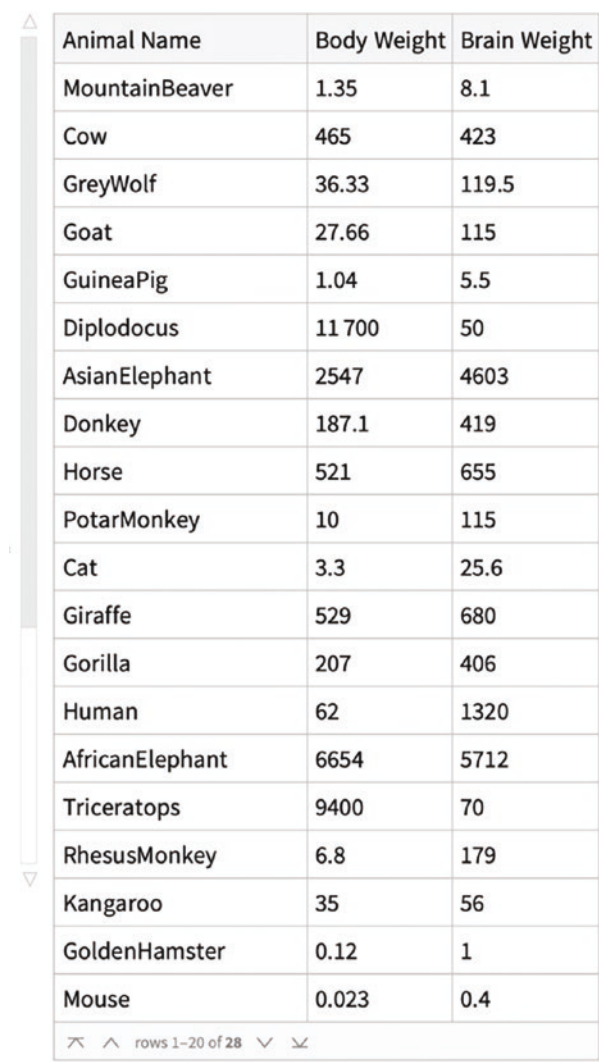
rows 1-4 of 28

**Figure 3-48.** Column 1 and row 1 suppressed

You can add headers to each column in the new dataset with the Query command. To rename the columns, the same procedure is applied; the new names would be ruled to the old names—that is, “New name” → “Animal Name,” as shown in Figure 3-49.

```
In[125]:= Query[All,<|"Animal Name"->1,"Body Weight"->2,"Brain
Weight"->3|>]@Dataset[AnimalData]
(*for display motives we put row 7 to 9,use All for the whole data set*)
(*or "symbol_of_the_dataset"[All,<|"Animal Name"-> 1,"Body Weight"->2,
"Brain Weight"->3|>]*)
Out[125]=
```





Animal Name	Body Weight	Brain Weight
MountainBeaver	1.35	8.1
Cow	465	423
GreyWolf	36.33	119.5
Goat	27.66	115
GuineaPig	1.04	5.5
Diplodocus	11 700	50
AsianElephant	2547	4603
Donkey	187.1	419
Horse	521	655
PotarMonkey	10	115
Cat	3.3	25.6
Giraffe	529	680
Gorilla	207	406
Human	62	1320
AfricanElephant	6654	5712
Triceratops	9400	70
RhesusMonkey	6.8	179
Kangaroo	35	56
GoldenHamster	0.12	1
Mouse	0.023	0.4

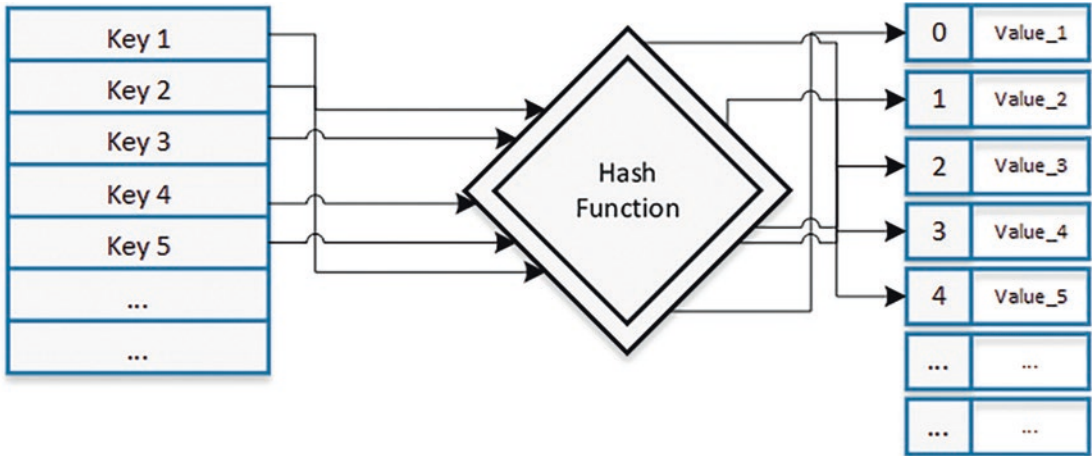
rows 1-20 of 28

**Figure 3-49.** *Animal dataset with added column headers*

## Generalization of Hash Tables

A hash table is an associative data structure that allows data storage and, in turn, the rapid retrieval of elements (values) from objects called keys. Hash tables can be implemented inside arrays, where the main components are the key and the value. The way to search for an element in the array is by using a hash function, which maps the keys to the pairs of values and gives you the place where it is in the array (index).

In other words, the hash function searches for a certain key, evaluates that key, and returns an index. This process is known as hashing. Figure 3-50 shows a representative schema of a hash table.



**Figure 3-50.** *Graphic representation of a hash table*

Inside the hash table, the number of keys and values can go on and on, which is one of the reasons hash tables are very useful; they can store large amounts of information. Inside the Wolfram Language, associations can represent hash tables. Primarily, this is because associations are an abstract data structure with fundamental components such as keys and values, just like a hash table. This combines the structure of an associative array and an indexed list, more like a nest of hash arrays. With the crucial property that associations are immutable, each association-type object is unique and the reference to one association has no link to another, even though they are referenced to the same symbol.

Other special commands are available. Let's first create an association. Nested associations are defined as associations that have associations within them—in other words, a key that points to a bucket of values that correspond to keys that have other values inside (see Figure 3-51).

```
In[126]:= Asc=<|"User"->
<|"Edgar"-> <|"id"->01, "Parameters"-><|"Active"->True,"Region"->
"LA","Internet Traffic"->"1 GB"|>>,
<|"Any"-><|"id"->02,"Parameters"-><|"Active"->False,"Region"->
"MX","Internet Traffic"->"3 GB"|>>
```

```
|>|>|>;  
Dataset[%]  
Out[127]=
```

		id	Parameters		
			Active	Region	Internet Traffic
User	Edgar	1	True	LA	1 GB
	Anya	2	False	MX	3 GB

**Figure 3-51.** Nested associations in the dataset format

Executing operations like accessing items, updating values, and deleting is supported by the commands associated with keys and values. Remember that Keys returns the keys of the association and Values the values. Keys only work at the surface level inside a nested association, as seen in the following code.

```
In[128]:= Keys[Asc]  
Out[128]= {User}
```

Applying the Keys command returns only the key user. The Keys command needs to be applied to deeper levels to see the keys inside a nested association, which is achieved with Map by specifying the sublevel only.

```
In[129]:= Map[Keys,Asc,#]&/@{{0},{1},{2}}//Column  
Out[129]= {User}  
<|User->{Edgar,Anya}|>  
<|User-><|Edgar->{id,Parameters},Any->{id,Parameters}|>|>
```

As seen on the surface level (0), the key is User. The next sublevel has the keys Edgar and Anya, and the last level has the keys ID and parameters for each of the keys Edgar and Anya. MapIndexed lets you look inside the whole association and apply Keys to sublevels to show the predecessors of the keys.

```

In[130]:=
Print["Level 0: "<>ToString@MapIndexed[Keys,Asc,{0}]]
Print["Level 1: "<>ToString@MapIndexed[Keys,Asc,{1}]]
Print["Level 2: "<>ToString@MapIndexed[Keys,Asc,{2}]]
Out[130]=
Level 0: {{}[User]}
Level 1: <|User -> {{Key[User]}[Edgar], {Key[User]}[Anya]}|>
Level 2: <|User -> <|Edgar -> {{Key[User], Key[Edgar]}[id], {Key[User],
Key[Edgar]}[Parameters]}, Anya -> {{Key[User], Key[Anya]}[id], {Key[User],
Key[Anya]}[Parameters]}|>|>

```

At level 0, only the User key exists, and the predecessor is {}. At level 1, the User predecessor and the Edgar and Anya keys are values of the User key. At level 2, the predecessor keys are Edgar/Anya and User for the ID and Parameters keys. In other words, the expression {Key[User], Key[Anya]}[id] means that ID corresponds to the Anya key and Anya to the User key, and so on. This is also useful because it means that access to a value or values of a key is done with the operator form applied to the association specifying the keys.

```

In[133]:= Asc["User"]["Edgar"]["id"](*{Key[User],Key[Anya]}[id],*)
Out[133]= 1

```

As shown, you get the value that corresponds to the ID inside Edgar inside User key. To see a graphical representation of the previous expression, you can use MapIndexed to label the positions of the keys and dataset applied, for example, in sublevel 4 (see Figure 3-52).

```

In[134]:= Dataset@MapIndexed[Framed[Labeled[#2,#1],FrameMargins->0,
RoundingRadius->5]&,Asc,{4}] (*Try changin the number to see how the
expression changes*)
Out[134]=

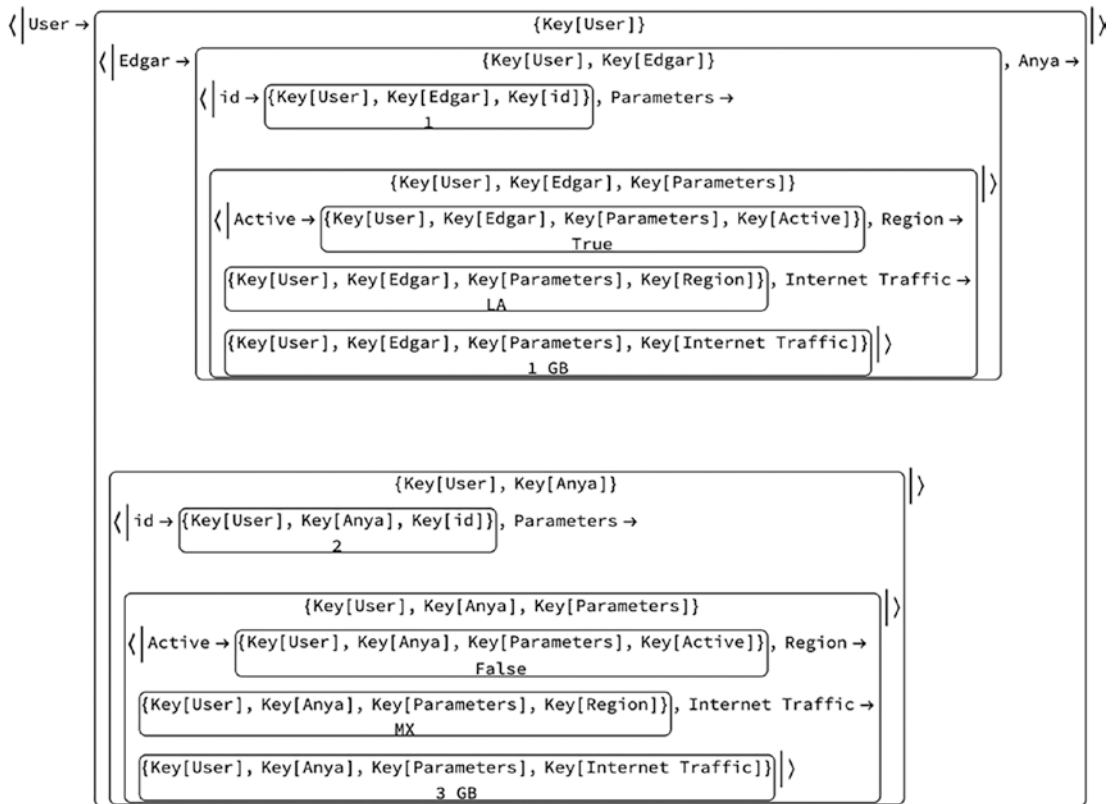
```

		id	Parameters	
User	Edgar	1	Active	{Key[User], Key[Edgar], Key[Parameters], Key[Active]} True
			Region	{Key[User], Key[Edgar], Key[Parameters], Key[Region]} "LA"
			Internet Traffic	{Key[User], Key[Edgar], Key[Parameters], Key[Internet Traffic]} "1 GB"
	Anya	2	Active	{Key[User], Key[Anya], Key[Parameters], Key[Active]} False
			Region	{Key[User], Key[Anya], Key[Parameters], Key[Region]} "MX"
			Internet Traffic	{Key[User], Key[Anya], Key[Parameters], Key[Internet Traffic]} "3 GB"

**Figure 3-52.** Dataset representation marking the keys inside the nested association

Each box contains the values of the predecessor key. This is why 1 GB corresponds to {Key[User],Key[Edgar],Key[Parameters],Key[Internet Traffic]}. To see the whole expression, the level of specification is Infinity (see Figure 3-53).

```
In[135]:=MapIndexed[Framed[Labeled[#2,#1,ImageMargins->0,Spacings->0],
FrameMargins->0,RoundingRadius->5]&,Asc,Infinity]
Out[135]=
```



**Figure 3-53.** Framed levels of the keys in a nested association

Values use the same approach as with Keys. To test if a key exists, use `KeyExistsQ`; this returns true if the key exists. Otherwise, it is false. To test inside deeper levels, use `Map`.

```
In[136]:= {KeyExistsQ[Asc, "User"], Map[KeyExistsQ["Anya"], Asc, {1}], Map
[KeyExistsQ["Anya"], Asc, {2}]}
Out[136]= {True, <|User->True|>, <|User-><|Edgar->False, Anya->False|>|>}
```

Another way to test whether a key in a particular form exists inside an association, use `KeyMemberQ`—for example, if there is a string pattern key.

```
In[137]:= KeyMemberQ[Asc["User"]["Anya"], _String]
Out[137]= True
```

To test if a value exists given a key, use `Lookup`.

```
In[138]:= Lookup[Asc["User"]["Anya"], "Parameters"]
Out[138]= <|Active->False, Region->MX, Internet Traffic->3 GB|>
```

To select a key based on criteria, use `KeySelect`.

```
In[139]:= KeySelect[Asc["User"]["Anya"],StringQ]
Out[139]= <|id->2,Parameters-><|Active->False,Region->MX,Internet
Traffic->3 GB|>|>
```

Or use `KeyTake` to grab a particular key.

```
In[140]:= KeyTake[Asc["User"]["Anya"]["Parameters"],{"Region","Internet
Traffic"}]
Out[140]= <|Region->MX,Internet Traffic->3 GB|>
```

To remove a key, use `KeyDrop`.

```
In[141]:= KeyDrop[Asc["User"],"Edgar"]
Out[141]= <|Anya-><|id->2,Parameters-><|Active->False,Region->MX,Internet
Traffic->3 GB|>|>|>
```

To assign a new value, the value associated with the key is assigned with the new value

```
In[142]:= Asc["User"]["Edgar"]["Parameters"]["Region"]="CZ"
Out[142]= CZ
```

Passing this into a dataset, you can look for the new assigned value (see Figure 3-54).

```
In[143]:= Dataset[Asc]
Out[143]=
```

		id	Parameters		
			Active	Region	Internet Traffic
User	Edgar	1	True	CZ	1 GB
	Anya	2	False	MX	3 GB

**Figure 3-54.** Dataset with the region value changed to CZ

To add a key and a value to the association, you can insert the new expression by specifying the position to insert it with the key (see Figure 3-55).

```
In[144]:= Insert[Asc["User"], "Alexandra" -> <|"id" -> 0, "Parameters" -> <|"Active" -> False, "Region" -> "RS", "Internet Traffic" -> "12 GB"|>>, Key["Edgar"]]/Dataset
Out[144]=
```

	id	Parameters		
		Active	Region	Internet Traffic
Alexandra	0	False	RS	12 GB
Edgar	1	True	CZ	1 GB
Anya	2	False	MX	3 GB

**Figure 3-55.** *New row added by the key position*

## Summary

This chapter continued to build upon the list operations introduced in Chapter 2. You explored the unique syntax of pure functions in the Wolfram Language and delved into several methods for creating indexed tables and associations. Additionally, you transitioned to the powerful capabilities of datasets, which provide a structured and organized way to handle and analyze data. The chapter wrapped up by providing insights into the essential components of associations and key-value management.



## CHAPTER 4

# Import and Export

This chapter reviews the import and export of data, including the relevant Wolfram Language commands and the import and export formats that Mathematica supports. Experimental data can come from different sources; the way to process this external data is to import it through Wolfram Language. Data that has been calculated or obtained externally can be transferred to Mathematica and exported for use on other platforms. However, Mathematica has tools to handle different data types (numbers, text, audio, graphics, and images). This chapter focuses on working with numerical and categorical data, the most frequently used data types for analysis.

Importing data from multiple sources into Mathematica allows you to load data into a notebook for analysis. The Wolfram Language supports numerous import formats; to see which are supported, type the dollar symbol (\$) accompanied by the ImportFormats command. Currently, Mathematica supports 256 file formats. As shown in the following code, new formats have been added and updated since the last version of this book.

```
In[1]:= Short[$ImportFormats,4](* Length[$ImportFormats] --> 256 formats*)
Out[1]//Short= {3DS,7z,AC,ACO,Affymetrix,AgilentMicroarray,AIFF,ApacheLog,
ArcGRID,ASC,ASE,AU,AVI,Base64,BDF,Binary,BioImageFormat,Bit,BLEND,BMP,
<<216>>,WAV,Wave64,WDX,WebP,WL,WLNet,WMLF,WXF,X3D,XBM,XGL,XHTML,XHTMLMathML,
XLS,XLSX,XML,XPORT,XYZ,ZIP,ZSTD}
```

There are a lot of formats in the list, including audio, image, and text. But let's focus on the text-based formats. To import any file, the Import command is used. Import receives two arguments: the file's path and options. Options can vary between file format, elements, and other types of objects in Mathematica, like cloud and local. To select a file path, head to the toolbar and then to Insert ► File Path. A file explorer should appear; search the file you would like to import and select it. The path is enclosed in apostrophes like a string.

Another option is the named file in the Insert menu. In contrast to File Path, the File option introduces the file's contents directly without receiving prior formatting from Mathematica. File is better suited for importing notebooks or other Wolfram formats.

---

**Note** The next series of imported files are included in the source code. The files are located in the host Desktop folder for ease of use.

---

Let's look at transferring a simple text file. First, select the HelloWorld.txt file path using the Import command.

```
In[2]:= Import["/Users/macosex/Desktop/Hello_World.txt"]  
Out[2]= Hello world!
```

---

**Note** Based on your operating system, the file path shows forward slashes (Linux, macOS) or back slashes (Windows file system delimiter).

---

You have imported your first file. Mathematica recognizes it based on the file extension and then imports it automatically. If you import a file with no file extension but you know the type of format used in the file, you can choose the proper format as an option.

```
In[3]:= Import["/Users/macosex/Desktop/Hello_World.txt", "Text"]  
Out[3]= Hello world!
```

## Importing Files

Importing simple text files is easy and intuitive. However, based on the type of file you want to import, the options and format to display the data inside Mathematica can vary.

## CSV and TSV Files

This section focuses on how to import files into Mathematica. The examples work with comma-separated value (CSV) files, tab-separated value (TSV) files, and Excel spreadsheet-style files. CSV and TSV files are files that include text and numeric values. In CSV files, fields are separated by a comma; each row is one line record. Meanwhile, in TSV files, each record is separated with a tab space.

With Import, you can import TSV or CSV files with the .tsv or .csv file extension, respectively. Let's first import a regular CSV file by introducing the file path and then the CSV option.

```
In[4]:= Import["/Users/macosex/Desktop/Grocery_List.csv", "CSV"]
Out[4]= {{id,grocery item,price,sold items,sales per day},{1,milk,4$,4,4 Jun 2019},{2,butter,3$,2,6 Jun 2019},{3,garlic,2$,1,7 Jun 2019},{4,apple,2$,4,1 Jun 2019},{5,orange,3$,5,8 Jun 2019},{6,orange juice,5$,2,8 Jun 2019},{7,cheese,5$,2,6 Jun 2019},{8,cookies,2$,5,9 Jun 2019},{9,grapes,4$,3,21 Jun 2019},{10,potatoe,2$,5,26 Jun 2019}}
```

Now that the contents of the file are imported, depending on the format of the contents, the data is presented as a nested list or not. The elements of the nested list represent rows, and the elements of the whole list represent columns.

When importing data, parts of the data can be imported—that is, if you only need a row or a column.

```
In[5]:= Import["/Users/macosex/Desktop/Grocery_List.csv", {"Data", 5;;10}]
Out[5]= {{4,apple,2$,4,1 Jun 2019},{5,orange,3$,5,8 Jun 2019},{6,orange juice,5$,2,8 Jun 2019},{7,cheese,5$,2,6 Jun 2019},{8,cookies,2$,5,9 Jun 2019},{9,grapes,4$,3,21 Jun 2019}}
```

The previous example imported data from row 5 to row 10.

You can use the following form when you are only interested in single values.

```
In[6]:= Import["/Users/macosex/Desktop/Grocery_List.csv", {"Data", 6, 2}]
Out[6]= orange
```

Depending on the maximum bytes of the expression, Mathematica truncates the imported data and shows you a suggestion box of a simplified version of the whole data. To see the maximum byte size, go to Edit ► Advanced tab, and in “Maximum output size before truncation,” enter the new number of bytes before truncation. This preference applies to every output expression in Mathematica.

Let's use the same approach to import TSV files. With the short command, you can show a part of the data, just in case the data is extensive.

```
In[7]:= Short[Import["/Users/macosex/Desktop/Color_table.tsv", "TSV"]]
(*Rest, to view the remain*)
Out[7]//Short= {{number,color},{1,red},{<<7>>},{9,magenta},{10,brown}}
```

Consequently, in the result, a seven appears among the elements of the imported file. This result happens because the file contains seven elements that are not visible. Now that you have learned how to import CSV and TSV files, you can display the imported data in table format using Grid or TableForm.

```
In[8]:= Import["/Users/macosex/Desktop/Grocery_List.csv","CSV"];
Grid[%]
Out[9]=
```

id	grocery item	price	sold items	sales per day
1	milk	4\$	4	4 Jun 2019
2	butter	3\$	2	6 Jun 2019
3	garlic	2\$	1	7 Jun 2019
4	apple	2\$	4	1 Jun 2019
5	orange	3\$	5	8 Jun 2019
6	orange juice	5\$	2	8 Jun 2019
7	cheese	5\$	2	6 Jun 2019
8	cookies	2\$	5	9 Jun 2019
9	grapes	4\$	3	21 Jun 2019
10	potato	2\$	5	26 Jun 2019

Once you have imported the file, data can be treated as a list or any other structure inside the notebook. Parts of the data are named after the imported data, and the contents can now be extracted, as discussed in later chapters.

## XLSX Files

The following example shows how to import data, display data as a spreadsheet, and transform it into a dataset. Let’s use the XLSX grocery list file rather than the CSV file for exemplification purposes. To start, you need first to import the data. To start, you need first to import the data.

```
In[10]:= path="/Users/macosex/Desktop/Grocery_List.xlsx";
Import[path,"Data"]
Out[11]= {{{id,grocery item, price, sold items,sales per day},{1.,milk,4
$,4.,4-Jun-2019},{2.,butter,3$,2.,6-Jun-2020},{3.,garlic,2
$,1.,7-Jun-2021},{4.,apple,2 $,4.,1-Jun-2022},{5.,orange,3
```

```
{5.,8-Jun-2023},{6.,orange juice,5 $,2.,8-Jun-2024},{7.,cheese,5 $,2.,6-Jun-2025},{8.,cookies,2 $,5.,9-Jun-2026},{9.,grapes,4 $,3.,21-Jun-2027},{10.,potatoe,2 $,5.,26-Jun-2028}}}
```

As can be seen, the imported data appears as a nested list because Excel files can have multiple sheets inside a file. For this case, you have only one sheet. To see the number of sheets and the name of the sheets, use `SheetCount` and `Sheets`, respectively.

```
In[12]:= Import[path,#]&@{"SheetCount","Sheets"}
Out[12]= {1,{Grocery_List}}
```

To show data as a spreadsheet, you use the `TableView` command (see Figure 4-1). The following format is used as an option to select a sheet: {"Data," # of sheet}. To select a character encoding, use the `CharacterEncoding` option. Also, custom rows or columns can be imported, preserving the format: {"Data," # of the sheet, # row, # column}.

```
In[13]:= TableView[Import[path,{"Data",1},CharacterEncoding->"UTF-8"]]
Out[13]=
```

	1	2	3	4	5
1	id	grocery	price	sold	sales per day
2		1. milk	4 \$		4. 4 - Jun - 2019
3		2. butter	3 \$		2. 6 - Jun - 2020
4		3. garlic	2 \$		1. 7 - Jun - 2021
5		4. apple	2 \$		4. 1 - Jun - 2022
6		5. orange	3 \$		5. 8 - Jun - 2023
7		6. orange	5 \$		2. 8 - Jun - 2024
8		7. cheese	5 \$		2. 6 - Jun - 2025
9		8. cookies	2 \$		5. 9 - Jun - 2026
10		9. grapes	4 \$		3. 21 - Jun - 2027
11		10. potatoe	2 \$		5. 26 - Jun - 2028

**Figure 4-1.** Spreadsheet view with `TableView` command

---

**Note** With "Data", import the data as a nested list.

---

You can now see the data in spreadsheet format. Now, with TableView, you can view the data like in spreadsheet software, with selection tools, scrollbars, and text editing of the contents. However, one of the downsides is that with TableView, you cannot directly access the file’s contents; neither can calculations be performed. To do the latter, you can transform it into a dataset.

You can convert data into a dataset for better handling in Mathematica. By typing the “Dataset” as the option instead of “Data”, the imported file becomes a dataset but without headers (see Figure 4-2). To add the headers, use the HeaderLines option and choose the specification of the header by row or column type `HeadLines → {# row, # column}`. The file used is Grocery List 2.xlsx.

```
In[14]:=file="/Users/macosex/Desktop/Grocery_List_2.xlsx";Import[file,
{"Dataset",1},HeaderLines->1]
Out[14]=
```

id	grocery item	price	sold items
1.0	milk	4\$	4.0
2.0	butter	3\$	
3.0	garlic	2\$	1.0
4.0	apple	2\$	4.0
5.0	orange		5.0
6.0	orange juice	5\$	2.0
7.0	cheese	5\$	2.0
8.0	cookies		5.0
9.0	grapes	4\$	
10.0	potato	2\$	5.0

**Figure 4-2.** *Incomplete Grocery List dataset*

You have imported incomplete data. EmptyField is implemented as a rule of transformation to treat empty spaces. If the data has empty spaces and no rule is expressed, the spaces are treated as empty strings. Figure 4-3 shows the output.

```
In[15]:= Import[file,{"Dataset",1},"EmptyField"->"NaN",HeaderLines->1]
Out[15]=
```

id	grocery item	price	sold items
1.0	milk	4\$	4.0
2.0	butter	3\$	NaN
3.0	garlic	2\$	1.0
4.0	apple	2\$	4.0
5.0	orange	NaN	5.0
6.0	orange juice	5\$	2.0
7.0	cheese	5\$	2.0
8.0	cookies	NaN	5.0
9.0	grapes	4\$	NaN
10.0	potato	2\$	5.0

**Figure 4-3.** *NaN-filled dataset*

## JSON Files

The JavaScript Object Notation (JSON) file extension is a data representation file. JSON files store data as an ordered list of values, and a collection of value pairs constitutes each list. To import a JSON file, specify the two options: JSON or RawJSON.

```
In[16]:=json=Import["/Users/macosex/Desktop/Sports_cars.json","JSON"]
```

```
Out[16]=
```

```
{ {Model->Enzo Ferrari,Year->2002,Cylinders->12,Horsepower HP->660,Weight Kg->1255}, {Model->Koenigsegg CCX,Year->2000,Cylinders->8,Horsepower HP->806,Weight Kg->1180}, {Model->Pagani Zonda,Year->2002,Cylinders->12,Horsepower HP->558,Weight Kg->1250}, {Model->McLaren Senna,Year->2019,Cylinders->8,Horsepower HP->800,Weight Kg->1309}, {Model->McLaren 675 LT,Year->2015,Cylinders->8,Horsepower HP->675,Weight Kg->1230}, {Model->Bugatti Veyron,Year->2006,Cylinders->16,Horsepower HP->1001,Weight Kg->1881}, {Model->Audi R8 Spyder,Year->2010,Cylinders->10,Horsepower HP->525,Weight Kg->1795}, {Model->Aston Martin Vantage,Year->2009,Cylinders->8,Horsepower HP->926,Weight Kg->1705}, {Model->Maserati Gran Turismo,Year->2010,Cylinders->8,Horsepower HP->405,Weight Kg->1955}, {Model->Lamborghini Aventador S,Year->2017,Cylinders->12,Horsepower HP->740,Weight Kg->1740} }
```

Given the nature of the JSON file structure, Mathematica recognizes each structure and interprets each key to its values when importing them. As you saw in the previous output, keys correspond to Model, Year, Cylinders, Horsepower, and Weight, and each key has its values. Everything said so far explains that all records are in a nested list. This outcome leads you to conclude that if you want to present it in a dataset, you cannot directly apply Association, and Association suppresses repeated keys. You must create an association for each record since it is a nested list, which you achieve with Map, specifying the depth level of the Association command. This is shown in the following code.

```
In[17]:= Map[Association,Json,1]
Out[17]= {<|Model->Enzo Ferrari,Year->2002,Cylinders->12,Horsepower HP->
660,Weight Kg->1255|>,<|Model->Koenigsegg CCX,Year->2000,Cylinders->8,
Horsepower HP->806,Weight Kg->1180|>,<|Model->Pagani Zonda,Year->2002,
Cylinders->12,Horsepower HP->558,Weight Kg->1250|>,<|Model->
McLaren Senna,Year->2019,Cylinders->8,Horsepower HP->800,Weight Kg->
1309|>,<|Model->McLaren 675 LT,Year->2015,Cylinders->8,Horsepower HP->675,
Weight Kg->1230|>,<|Model->Bugatti Veyron,Year->2006,Cylinders->16,
Horsepower HP->1001,Weight Kg->1881|>,<|Model->Audi R8 Spyder ,Year->2010,
Cylinders->10,Horsepower HP->525,Weight Kg->1795|>,<|Model->Aston Martin
Vantage,Year->2009,Cylinders->8,Horsepower HP->926,Weight Kg->1705|>,
<|Model->Maserati Gran Turismo,Year->2010,Cylinders->8,Horsepower HP-
>405,Weight Kg->1955|>,<|Model->Lamborghini Aventador S,Year->2017,
Cylinders->12,Horsepower HP->740,Weight Kg->1740|>}
```

You already have each record as an association, and now you can convert it to a dataset, as shown in Figure 4-4.

```
In[18]:= Dataset[%]
Out[18]=
```



Model	Year	Cylinders	Horsepower HP	Weight Kg
Enzo Ferrari	2002	12	660	1255
Koenigsegg CCX	2000	8	806	1180
Pagani Zonda	2002	12	558	1250
McLaren Senna	2019	8	800	1309
McLaren 675 LT	2015	8	675	1230
Bugatti Veyron	2006	16	1001	1881
Audi R8 Spyder	2010	10	525	1795
Aston Martin Vantage	2009	8	926	1705
Maserati Gran Turismo	2010	8	405	1955
Lamborghini Aventador S	2017	12	740	1740

**Figure 4-4.** Cars dataset

You can now handle a JSON file as a dataset. However, there is another way to do it without requiring as much calculation as before. When importing the file, you must import it as RawJson because, with RawJson, the Wolfram Language identifies and imports each record as a list of associations rather than a sole nested list, as shown here. This reason is because of the nature of the key and value of the JSON file extension.

```
In[19]:= Import["/Users/macosex/Desktop/Sports_cars.json","RawJSON"]
Out[19]=
{<|Model->Enzo Ferrari,Year->2002,Cylinders->12,Horsepower HP->660,Weight
Kg->1255|>,<|Model->Koenigsegg CCX,Year->2000,Cylinders->8,Horsepower HP->
806,Weight Kg->1180|>,<|Model->Pagani Zonda,Year->2002,Cylinders->12,
Horsepower HP->558,Weight Kg->1250|>,<|Model->McLaren Senna,Year->2019,
Cylinders->8,Horsepower HP->800,Weight Kg->1309|>,<|Model->McLaren 675
LT,Year->2015,Cylinders->8,Horsepower HP->675,Weight Kg->1230|>,<|Model->
Bugatti Veyron,Year->2006,Cylinders->16, Horsepower HP->1001,Weight Kg->
1881|>,<|Model->Audi R8 Spyder ,Year->2010,Cylinders->10,Horsepower HP->
525,Weight Kg->1795|>,<|Model->Aston Martin Vantage,Year->2009,
Cylinders->8,Horsepower HP->926,Weight Kg->1705|>,<|Model->Maserati Gran
Turismo,Year->2010,Cylinders->8,Horsepower HP->405,Weight Kg->1955|>,
<|Model->Lamborghini Aventador S,Year->2017,Cylinders->12,Horsepower
HP->740,Weight Kg->1740|>}
```

The file is imported as an association in each record, and you can convert it into a dataset.

```
In[20]:=Cars=Dataset[%];
```

As a complement, once the data is imported, you can perform operations on the dataset, such as ordering the models by year from low to high.

```
In[21]:=Cars[SortBy[#Year&]];
```

---

**Note** The previous example is also possible using the query command. (Query [SortBy[#Year &]][Cars]).

---

## Web Data

On the other hand, web data is also supported with Import. Instead of inserting the file path, the URL site is inserted as the argument of the Import command. The next example imports a simple text file from the National Oceanic and Atmospheric Administration (NOAA). The text file contains the list of country codes used for the Integrated Global Radiosonde Archive (IGRA). The parent directory where files are located is <https://www1.ncdc.noaa.gov/pub/data/igra/>, but let's only import the country list file. You need an Internet connection to make this work.

```
In[22]:=Short[Import["https://www1.ncdc.noaa.gov/pub/data/igra/igra2-
country-list.txt","HTML"]]
Out[22]//Short= AC Antigua and Barbuda AE United Arab Emirates AF ... WS
Samoa YM Yemen ZA Zambia ZI Zimbabwe ZZ Ocean
```

The file is a plain text, but you can change how the data is imported by inserting a file format as an option. You can import it as a CSV file, for instance.

```
In[23]:=Short[Import["https://www1.ncdc.noaa.gov/pub/data/igra/igra2-
country-list.txt","CSV"]]
Out[23]//Short= {{AC Antigua and Barbuda},{AE United Arab
Emirates},<<215>>,{ZI Zimbabwe},{ZZ Ocean}}
```

This is useful when you try to make computations with the data imported. Alternatively, you can use URL commands to check the status of an online file and then download it. To check the status of the online file, use `URLRead`. When the file is online, you should get an HTTP response object like the one shown in Figure 4-5. You can even perform this approach before importing data, ensuring the content is available online.

```
In[24]:= URLRead["https://www1.ncdc.noaa.gov/pub/data/igra/igra2-country-
list.txt"]
Out[24]=
```



```
HTTPResponse [ 200 Status: OK
Content type: text/plain ]
```

**Figure 4-5.** *HTTPResponse object of the URL entered*

Now that you know the status, you can download the data file with `URLDownload`.

```
In[25]:= URLDownload["https://www1.ncdc.noaa.gov/pub/data/igra/igra2-
country-list.txt"]
Out[25]=
```

You should get a file object with the file's location (see Figure 4-6), the name, and the extension; in this case, it is in a temporary folder.



```
File [ /private/var/folders/zs/hxtbpjpd5xb0krb6581764xm0000gn/T/igra2-country-list-b7add4af-0a18-4cea-bbcd-3ec04b94f363.txt >> ]
```

**Figure 4-6.** *File object with the locations of the file downloaded*

Click the double chevron icon to open the file in an external viewer.

## Semantic Import

So far, you have seen how to import files of different formats, but there is another tool called `SemanticImport` that allows you to import files semantically and returns a dataset as a result. Let's look at a simple example with the CSV file.

```
In[26]:= sImptr=SemanticImport["/Users/macosex/Desktop/Grocery_List.csv"]
Out[26]=
```

Figure 4-7 shows that when you use semantic import Mathematica, it imports the data in the form of a dataset, and when it does this, it recognizes some quantities.

id	grocery item	price	sold items	sales per day
1	milk	\$4	4	Tue 4 Jun 2019
2	butter	\$3	2	Thu 6 Jun 2019
3	garlic	\$2	1	Fri 7 Jun 2019
4	apple	\$2	4	Sat 1 Jun 2019
5	orange	\$3	5	Sat 8 Jun 2019
6	orange juice	\$5	2	Sat 8 Jun 2019
7	cheese	\$5	2	Thu 6 Jun 2019
8	cookies	\$2	5	Sun 9 Jun 2019
9	grapes	\$4	3	Fri 21 Jun 2019
10	potato	\$2	5	Wed 26 Jun 2019

**Figure 4-7.** *File imported as a dataset with SemanticImport*

These quantities correspond to the magnitude and its units, such as in the case of the elements of the column of price and sales per day. When dealing with quantities, the color of the elements changes; as you see in the dataset, the elements appear differently from the other contents because a semantic-type object now represents them. Semantic objects include quantities, entities, dates, and geolocation. In other words, they are interpretations made by the freeform interpreter related to the Wolfram Knowledgebase.

---

**Note** To check if the data is recognized as a quantity or semantic-type object, use `Normal[slmprt]`; you should see the entities colored differently.

---

In the case of imported data, there are two date-type objects, which you saw in the first chapter, and quantity type. It should be understood that to work with quantities, you must understand where they come from.

## Quantities

The `Quantity` command converts a magnitude with units to a quantity type to convert the magnitude with their respective units; the magnitude is entered first, followed by its units in string type. When you do this, Mathematica displays the autocomplete menu as on other occasions. The following example shows it.

```
In[27]:= Quantity[2,"USDollars"]
Out[27]= $2
```

Thus, it is transformed into a quantity type. When you hover over the result, an ad is displayed, marking that a result is already a unit. In this case, it is a unit of US dollars. Now, if you check the head of the expression, it shows that it is a type of quantity.

---

**Note** Quantities are shown in light brown color.

---

```
In[28]:= Quantity[2,"USDollars"]//Head
Out[28]= Quantity
```

You can also use the inline freeform input in the menu bar: Insert ► Inline Freeform Input. This input type is associated with the Wolfram Alpha search engine, so the inline freeform input transforms natural language into Wolfram Language input.

Inside the box, you'll find the magnitude and quantity written. One of the advantages of this type of input is that it allows for using natural language. The following example writes the amount of 77 min, which means 77 minutes. Figure 4-8 shows the input cell of the inline freeform input.

```
In[29]:=
```



**Figure 4-8.** Free inline freeform input for the quantity of 77 minutes

```
Out[29]= 77min
```

To run the code, click ENTER since it gives you a result. Some tabs appear where you can click a submenu or a checkmark. If you click the checkmark, it is to accept the interpretation made. If you believe that the interpretation is different, you can click the other option, which is alternate interpretations, and it shows a small pop-up where it lists different interpretations. Figure 4-9 show the pop-up for the example.



**Figure 4-9.** *Options for the quantity entered*

Once the interpretation is accepted, the result changes color and is a quantity-type object. And it can be used like any other quantity-type object.

When you have quantities, you cannot make operations between numbers; quantities are already different types. For these, there are two options: convert the data to quantities or extract the magnitude of a quantity. The `QuantityMagnitude` command is used to extract the magnitude. Make sure to copy the entity (light brown output), not the pure text 77 min.

```
In[30]:= {QuantityMagnitude[77 min],Head[%]}
Out[30]= {77,Quantity}
```

You have already extracted the magnitude, and it is already an integer. In the supposed case of wanting the units, the `QuantityUnit` command extracts the units.

```
In[31]:= QuantityUnit[77 min]
Out[31]= Minutes
```

## Datasets with Quantities

Another aspect to emphasize: To carry out operations, the concept of performing arithmetic operations among physical quantities is maintained; otherwise, the operation is not possible, and you get an error in which the units do not agree. When you carry out an operation between quantities, the result is also of the quantity type.

```
In[32]:= {77min-77min,77min+77min,77min*77min,77min/77min,77min*3m}
Out[32]= {0min,154min,5929(min)^2,1,231m min}
```

This example shows how the results are of type quantity. Except for the division, it is already a quotient between the same units. The last one is 231 meters per minute.

Returning to the imported data, you can extract the data from the price column, as shown in Figure 4-10.

```
In[33]:= sImprt[[All,"price"]]
Out[33]=
```

\$4	\$3	\$2	\$2	\$3
\$5	\$5	\$2	\$4	\$2

**Figure 4-10.** Price column

If you want to have them in a list, you must use the Normal command.

```
In[34]:= Normal[%]
Out[34]= {$ 4,$ 3,$ 2,$ 2,$ 3,$ 5,$ 5,$ 2,$ 4,$ 2}
```

The result is the list but in quantity type. It is fair to say that you can do operations with quantities, but if what matters are the magnitudes, you can extract them. It's worth noting that working with magnitudes alone is generally faster and more efficient, which reduces the overhead or additional quantity processing. Unless a specific quantity is required, converting to pure numbers may be preferable.

Let's look at how.

```
In[35]:= QuantityMagnitude[#]&[%]
Out[35]= {4,3,2,2,3,5,5,2,4,2}
```

You are now working with only the magnitudes.

You can even work with dates and quantities, as shown in Figure 4-11, starting by displaying the ID of the products and the date they were sold.

```
In[36]:= sImprt[[All,{"id","sales per day"}]]
Out[36]=
```

id	sales per day
1	Tue 4 Jun 2019
2	Thu 6 Jun 2019
3	Fri 7 Jun 2019
4	Sat 1 Jun 2019
5	Sat 8 Jun 2019
6	Sat 8 Jun 2019
7	Thu 6 Jun 2019
8	Sun 9 Jun 2019
9	Fri 21 Jun 2019
10	Wed 26 Jun 2019

**Figure 4-11.** *ID and sales per day columns*

Having done this, you can extract the values and work directly with the date object types.

```
In[37]:= Normal[Values[%]]//InputForm
Out[37]//InputForm=
{{1, DateObject[{2019, 6, 4}, "Day"]},
 {2, DateObject[{2019, 6, 6}, "Day"]},
 {3, DateObject[{2019, 6, 7}, "Day"]},
 {4, DateObject[{2019, 6, 1}, "Day"]},
 {5, DateObject[{2019, 6, 8}, "Day"]},
 {6, DateObject[{2019, 6, 8}, "Day"]},
 {7, DateObject[{2019, 6, 6}, "Day"]},
 {8, DateObject[{2019, 6, 9}, "Day"]},
 {9, DateObject[{2019, 6, 21}, "Day"]},
 {10, DateObject[{2019, 6, 26}, "Day"]}}
```

Each value represents a date using DateObject, which is easily converted to numeric values using AbsoluteTime. It is handy for numerical operations involving dates, making the data handling more flexible and efficient.



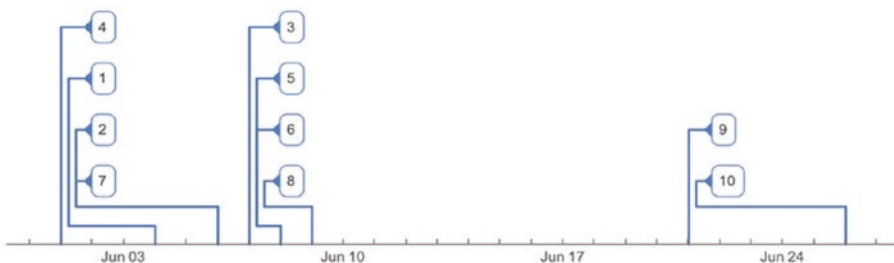
**Note** You should get the date object when testing the code instead of the pure word; here, the `InputForm` is used to avoid image conflicts.

Knowing this, you can make an association between the IDs of each product and when it was sold, applying the `Rule` command inside the nested list and creating the associations.

```
In[38]:= Association[Apply[Rule,%,1]]//InputForm
Out[38]//InputForm=
<|1 -> DateObject[{2019, 6, 4}, "Day"],
 2 -> DateObject[{2019, 6, 6}, "Day"],
 3 -> DateObject[{2019, 6, 7}, "Day"],
 4 -> DateObject[{2019, 6, 1}, "Day"],
 5 -> DateObject[{2019, 6, 8}, "Day"],
 6 -> DateObject[{2019, 6, 8}, "Day"],
 7 -> DateObject[{2019, 6, 6}, "Day"],
 8 -> DateObject[{2019, 6, 9}, "Day"],
 9 -> DateObject[{2019, 6, 21}, "Day"],
10 -> DateObject[{2019, 6, 26}, "Day"]|>
```

To illustrate this, create a visualization in a timeline, as shown in Figure 4-12, marking the product sold and the date of its sale.

```
In[39]:= TimelinePlot[%]
Out[39]=
```



**Figure 4-12.** Timeplot

The date of each grocery item sold is shown by ID. A tooltip shows the exact date when the cursor is passed over the number in the timeline.

The idea is that when you use `SemanticImport`, you can integrate different forms of the Wolfram Language and how you can use this to your advantage when importing data. Semantic import makes it possible to compare data with other selected data. `SemanticImport` provides you with tools to work among various types of semantic objects. What is essential to observe is that instead of importing standard text, you can import currency types, dates, and any magnitude with the respective unit, as in the previous examples. This allows that data to be associated with different commands within the Wolfram Language.

## Costume Import (Dealing with Large Datasets)

Having said all this about semantic import, you can import data and choose how each column in the imported file should be interpreted. However, based on the same idea that you saw earlier, with semantic import, you can also choose what data to import (e.g., if it is only one column or several), as illustrated in [Figure 4-13](#).

```
In[40]:= SemanticImport["/Users/macosex/Desktop/Grocery_List.csv",{"Integer",
"String","Currency","Real","Date"}]
```

```
Out[40]=
```

id	grocery item	price	sold items	sales per day
1	milk	\$ 4	4.0	Tue 4 Jun 2019
2	butter	\$ 3	2.0	Thu 6 Jun 2019
3	garlic	\$ 2	1.0	Fri 7 Jun 2019
4	apple	\$ 2	4.0	Sat 1 Jun 2019
5	orange	\$ 3	5.0	Sat 8 Jun 2019
6	orange juice	\$ 5	2.0	Sat 8 Jun 2019
7	cheese	\$ 5	2.0	Thu 6 Jun 2019
8	cookies	\$ 2	5.0	Sun 9 Jun 2019
9	grapes	\$ 4	3.0	Fri 21 Jun 2019
10	potato	\$ 2	5.0	Wed 26 Jun 2019

id	grocery item	price	sold items	sales per day
1	milk	\$4	4	Tue 4 Jun 2019
2	butter	\$3	2	Thu 6 Jun 2019
3	garlic	\$2	1	Fri 7 Jun 2019
4	apple	\$2	4	Sat 1 Jun 2019
5	orange	\$3	5	Sat 8 Jun 2019
6	orange juice	\$5	2	Sat 8 Jun 2019
7	cheese	\$5	2	Thu 6 Jun 2019
8	cookies	\$2	5	Sun 9 Jun 2019

**Figure 4-13.** Dataset with excluded rows

With this result, observe that the first column imported contains integers, the second contains text, the third contains a currency type quantity, the fourth contains a real number, and the last contains a date object. Having done this, it is possible in the same way that with spreadsheet files, you can import certain types of information in list form, either by column or by row. The following example imports rows 1 through 5.

```
In[41]:=SemanticImport["/Users/macosex/Desktop/Grocery_List.
csv",Automatic,"Rows"][[1;5]]//InputForm
Out[41]//InputForm= {{1, "milk", Quantity[4, "USDollars"], 4,
DateObject[{2019, 6, 4}, "Day"]}, {2, "butter", Quantity[3, "USDollars"],
2, DateObject[{2019, 6, 6}, "Day"]}, {3, "garlic", Quantity[2,
"USDollars"], 1, DateObject[{2019, 6, 7}, "Day"]}, {4, "apple", Quantity[2,
"USDollars"], 4, DateObject[{2019, 6, 1}, "Day"]}, {5, "orange",
Quantity[3, "USDollars"], 5, DateObject[{2019, 6, 8}, "Day"]}}
```

As indicated, columns can also be imported from columns 1 to 2.

```
In[42]:=SemanticImport["/Users/macosex/Desktop/Grocery_List.
csv",Automatic,"Columns"][[1;2]]
Out[42]= {{1,2,3,4,5,6,7,8,9,10},{milk,butter,garlic,apple,orange,orange ju
ice,cheese,cookies,grapes,potato}}
```

It is necessary to emphasize that if you want to exclude data, importing with the `ExcludedLines` statement is recommended. For example, exclude rows 9 and 10, remembering that the titles are in row 1, as shown in Figure 4-13.

```
In[43]:=SemanticImport["/Users/macosex/Desktop/Grocery_List.csv",ExcludedLines->{{10},{11}}]
Out[43]=
```

When working with large datasets, it’s crucial to manage memory usage. Review if your system can handle big sizes of data. If it’s too large to import at once, try importing it in smaller pieces and filtering/managing the data as needed. The following example effectively selects the first ten buildings (see Figure 4-14) from the buildings.dat dataset based on the specified condition using a pure function within Select.

```
In[44]:= SemanticImport["ExampleData/buildings.dat",<|"Name"->Automatic,"City"->Automatic,"Country"->Automatic,"Year"->Automatic|>,HeaderLines->1];
Select[%,#[[4]]<=2000&][[1;;10]]
Out[45]=
```

Name	City	Country	Year
Petronas Tower 1	Kuala Lumpur	Malaysia	1998
Petronas Tower 2	Kuala Lumpur	Malaysia	1998
Sears Tower	Chicago	United States	1974
Jin Mao Building	Shanghai	China	1999
CITIC Plaza	Guangzhou	China	1996
Shun Hing Square	Shenzhen	China	1996
Empire State Building	New York City	United States	1931
Central Plaza	Hong Kong	China	1992
Bank of China	Hong Kong	China	1989
Emirates Tower One	Dubai	United Arab Emirates	1999

**Figure 4-14.** Buildings dataset with selected rows

To filter the data dataset based on the condition that the Year column (index 4) is less than or equal to 2000. Then, use [[1;; 10]] to select the first ten elements from the filtered dataset, which are the first ten buildings that meet the condition.

# Export

Mathematica supports many formats; to view all supported formats, type \$ExportFormat.

```
In[46]:= Short[$ExportFormats,5]
Out[46]//Short= {3DS,AC,ACO,AIFF,ASE,AU,AVI,Base64,Binary,Bit,BLEND,BMP,
BREP,BSON,Byte,BYU,BZIP2,C,CDF,<<167>>,WDX,WebP,WL,WLNet,WMLF,WXF,X3D,XBM,
XGL,XHTML,XHTMLMathML,XLS,XLSX,XML,XPORT,XYZ,ZIP,ZPR,ZSTD}
```

Exporting data is carried out using the Export command. Export has the form  
Export["directory path," expr, "format"].

First, you need to set up a working directory. If not, the file is exported to the default Mathematica working directory. To see the working default directory, use Directory.

```
In[47]:= Directory[]
Out[47]= /Users/macosex
```

In this case, the default directory is the Desktop folder.

Two commands are key; one is SetDirectory, whose argument is the path of the new working directory, and the other is NotebookDirectory, which is the file's location.

First, let's set the new working directory to export files to the notebook location. Using the notebook directory as the argument on SetDirectory, you tell Mathematica that the new working directory is the location of the notebook in which you are currently working.

```
In[48]:= SetDirectory[NotebookDirectory[]]
Out[48]= /Users/macosex/Desktop
```

Now that you have set up a new directory, you can export data created in Mathematica. The next example exports a list of prime numbers from 1 to 10 as a table in a text file and a CSV file. An option applies as well as Import, but if the file extension is added, it is not compulsory to write the format option.

---

**Note** There is no restriction about whether to assign a name to the list of data or to create the data directly in the export.

---

```
In[49]:= mydata=Table[Prime[i],{i,1,10}];
{Export["New_File.txt",mydata,"Table"], Export["New_File.csv",mydata]}
Out[50]= {New_File.txt,New_File.csv}
```

The output generates the name of the new file exported. An alternative is manually entering the desired location of the file instead of setting a new working directory; in this case, Desktop was set as the new location.

```
In[51]:= Export["/Users/macosex/Desktop/New_File.TSV",mydata,"TSV"]
Out[51]= /Users/macosex/Desktop/New_File.TSV
```

Now that you have exported the data into a new location, the output is the full path of the new file. If you want to open the file from Mathematica, you can use `SystemOpen`. This command opens the operating system explorer.

```
In[52]:= SystemOpen["/Users/macosex/Desktop/New_File.TSV"]
```

`SystemOpen` lets you open the notebook directory folder to open other files inside the notebook directory.

```
In[53]:= SystemOpen[NotebookDirectory[]]
```

On the other hand, when dealing with tabular data, it can be exported as a spreadsheet. The next example export a tabular data structure and then export it into a spreadsheet format.

To create tabular data, let's use the `Table` command.

```
In[54]:=
tabD1=Table[i,{i,4}];
tabD2=SetPrecision[Table[i/11,{i,4}],3];
```

Now that you have a set of coordinates, you can export the data to different sheets by typing the reference name of the data into a list of options: {data\_sheet 1,data\_sheet 2, ...}

```
In[56]:= Export["Tabular_data.xls",{tabD1},{tabD2}]
Out[56]= Tabular_data.xls
```

By opening the file with a spreadsheet viewer, you should get that `TabD1` is in sheet 1 and `TabD2` is in sheet 2.

To customize the name of the sheets, you need to enter the names as a list of rules with the rule operator ( $\rightarrow$ ).

```
In[57]:= Export["Tabular_data_2.xls",{ "Page number 1"->tabD1,"Page number
2"->tabD2}]
Out[57]= Tabular_data_2.xls
```

If you open the file, now you should have two sheets with the names you have set.

In addition to this, there is the possibility to add the same data in a single spreadsheet. You only have to enclose the data you want in the same sheet in curly braces to do this.

```
In[58]:= Export["New_data.xls",Transpose[{tabD1,tabD2}]]
Out[58]= New_data.xls
```

After opening the file, you should see something like the following code.

```
In[59]:= Grid[Transpose[{tabD1,tabD2}]]
Out[59]=
1      0.0909
2      0.182
3      0.273
4      0.364
```

You can even export tables.

```
In[60]:= table1={{ "Dog", "Wolf"}, {"Cat", "Leopard"}, {"Pigeon", "Shark"}};
Export["Animal_table.xls",table1]
Out[61]= Animal_table.xls
```

## Other Formats

By advancing the topic, it is possible to export the data to simple formats such as TXT, DAT, CSV, and CSV. To do this, you only have to put the path of the file where you want it to be exported, along with the name of the new file, followed by the extension of the desired file. The second argument writes the data to be exported or the variable that contains the data. The third argument is what designates the format you want the data to import.

Let's look at the following example, which exports new data to text and DAT formats. In this case, you only write the file's name, which indicates that you want it to be exported to the working directory established earlier, corresponding to the notebook's directory.

```
In[62]:= newD=Table[{i+j,i*j},{i,1,5},{j,1,5}];
{Export["File_text.txt",newD,"Text"],Export["File_dat.dat",newD,"Table"]}
Out[63]= {File_text.txt,File_dat.dat}
```

It is advisable to pause for a moment. As shown in the earlier code, the Table format is used for the DAT file. This is because a Table is used so that the exported data becomes an expression in the Wolfram Language. After you have exported, verify that the files have been exported. Likewise, you can choose the format for a file. For example, instead of typing text, you export it in the TSV format.

```
In[64]:= Export["File_text.txt",newD,"TSV"]
Out[64]= File_text.txt
```

Similarly, you can export CSV and TSV files.

```
In[65]:={Export["File_csv.csv",newD,"CSV"],Export["File_tsv.
tsv",newD,"TSV"]}
Out[65]= {File_csv.csv,File_tsv.tsv}
```

It is possible to add titles to the columns of the data for when they are exported, either CSV or TSV.

```
In[66]:= Export["File_csv.csv",newD,"CSV",TableHeadings->{"column
1","column 2","column 3","column 4","column 5"}]
Out[66]= File_csv.csv
```

It is also possible to define a list of names for the columns as follows.

```
In[67]:= labels={"Coordinates 1","Coordinates 2","Coordindates
3","Coordinates 4","Coordindates 5"};Export["File_csv.csv",newD,"CSV",
TableHeadings->labels]
Out[67]= File_csv.csv
```

In the same way, you can export datasets to known formats. Let's use automobile braking distance statistics based on speed. For this, the data is loaded using the ExampleData command. Inside this, search "Statistics"; within that, search "CarStoppingDistances".

```
In[68]:= spData=ExampleData[{"Statistics","CarStoppingDistances"}]
Out[68]={{{4,2},{4,10},{7,4},{7,22},{8,16},{9,10},{10,18},{10,26},{10,34},
{11,17},{11,28},{12,14},{12,20},{12,24},{12,28},{13,26},{13,34},{13,34},
{13,46},{14,26},{14,36},{14,60},{14,80},{15,20},{15,26},{15,54},{16,32},
{16,40},{17,32},{17,40},{17,50},{18,42},{18,56},{18,76},{18,84},{19,36},
{19,46},{19,68},{20,32},{20,48},{20,52},{20,56},{20,64},{22,66},{23,54},
{24,70},{24,92},{24,93},{24,120},{25,85}}}
```



To get the dataset's columns and a description, add `Description` and `ColumnDescriptions`.

```
In[69]:= ExampleData[{"Statistics","CarStoppingDistances"},#]&@{"Description",
"ColumnDescriptions"}
Out[69]= {Car stopping distances as a function of speed.,{Speed in miles
per hour.,Stopping distance in feet.}}
```

Continuing the exploration, you see that the first numbers represent the speed in miles per hour, and the second numbers represent the distance in feet.

---

**Note** For more information, add properties as the second argument to `ExampleData`.

---

Moving forward in the exercise, you can add the column titles. This distinguishes each data type when you build the dataset (see Figure 4-15).

```
In[70]:= spDataset=Dataset[spData,Background->LightBlue][All,<|#1->1,#2->
2|>]&["Speed in miles per hours","Stopping distance in feet"]
Out[70]=
```

Speed in miles per hours	Stopping distance in feet
4	2
4	10
7	4
7	22
8	16
9	10
10	18
10	26
10	34
11	17
11	28
12	14
12	20
12	24
12	28
13	26
13	34
13	34
13	46
14	26

**Figure 4-15.** *CarStoppingDistances dataset*

You have finished the creation of the dataset. This data and the respective column titles can now be exported to a CSV format.

```
In[71]:= Export["Dataset_csv.csv",spDataset,"CSV"]
Out[71]= Dataset_csv.csv
```

If the export is successful, you should have a CSV file in the correct format. For the case of a TSV file, see the following form.

```
In[72]:= Export["Dataset_tsv.tsv", spDataset, "TSV"]
Out[72]= Dataset_tsv.tsv
```

## XLS and XLSX Formats

It is worth distinguishing that to export datasets to spreadsheet formats such as XLS or XLSX, you should work the dataset as a list since exporting the dataset directly would result in exporting associations in a single cell, and you are not interested in that. Regarding the second point, since you have the dataset, to extract the values, you use the Normal command, which converts the dataset into a normal expression, followed by extracting the values from the braces with Values.

```
In[73]:= Values@Normal@spDataset
Out[73]= {{4,2},{4,10},{7,4},{7,22},{8,16},{9,10},{10,18},{10,26},{10,34},
{11,17},{11,28},{12,14},{12,20},{12,24},{12,28},{13,26},{13,34},{13,34},
{13,46},{14,26},{14,36},{14,60},{14,80},{15,20},{15,26},{15,54},{16,32},
{16,40},{17,32},{17,40},{17,50},{18,42},{18,56},{18,76},{18,84},{19,36},
{19,46},{19,68},{20,32},{20,48},{20,52},{20,56},{20,64},{22,66},{23,54},
{24,70},{24,92},{24,93},{24,120},{25,85}}
```

Now that you have the data, you can add the column titles and export the extracted data from the dataset.

```
In[74]:= colTitles={"Speed in miles per hours","Stopping distance
in feet"};
```

To attach the two lists, let's use Prepend and assign the name `expData` to new values.

```
In[75]:= Short[expData=Prepend[%,colTitles],1]
Out[75]//Short= {{Speed in miles per hours,Stopping distance in feet},{4,2},
{4,10},<<45>>,{24,93},{24,120},{25,85}}
```

You do not define variables to put together this data list and titles. A percentage notation is used to simplify the code. Now that you have complete data, you can export it to an XLS or XLSX format.

```
In[76]:= Export["Stopping_distance_Dataset.xlsx",expData,"XLSX"]
Out[76]= Stopping_distance_Dataset.xlsx
```

If you verify the file, you should have something like the dataset created earlier.

## JSON Formats

It is also possible to export information to formats such as JSON. The following example creates a JSON structure from an association.

```
In[77]:= Association@{"Name"->"Ellis","Date of birth"->
"1990,01,04","Height"->"180 cm","Favorite color"->"Red","Hobbies"->"Soccer,
Pc gaming, Board games","Social networks"->"Twitter, Facebook"};
Export["File_json.json",%,"JSON"]
Out[78]= File_json.json
```

If you open the new JSON file, you see that it has a structure corresponding to a JSON file. It is the same process for the case where you have a nested list, although you can also use the “Rawjson” format when exporting. The idea is that you can export data to JSON formats from associations; as you have seen, the braces and values of an association can be any expression. This leads you to say that more associations can be added, and these can be exported. The vital thing to note is that given the nature of the JSON format of containing braces and values in pairs, it is possible to export data in JSON format from associations. Examining the case for when you have a dataset (see Figure 4-16), proceed as noted here.

```
In[79]:=Association@{"Name"->"Ellis","Date of birth"->
DateObject[{1990,01,04}], "Height"->Quantity[180,"Centimeters"], "Favorite
color"->"Red","Hobbies"->"Soccer, Pc gaming, Board games","Social networks"->
"Twitter, Facebook"};
user=Dataset[%]
Out[80]=
```

Name	Ellis
Date of birth	Thu 4 Jan 1990
Height	180 cm
Favorite color	Red
Hobbies	Soccer, Pc gaming, Board games
Social networks	Twitter, Facebook

**Figure 4-16.** *JSON file dataset*

The dataset is built, but in some cases, the dataset may contain quantities or other semantic objects, as in this case, the date and height. So, exporting them would be the same way as before but using the JSON option format, not Rawjson, since this does not allow exporting dataset objects. To use Rawjson, you must convert the semantic objects to strings or numbers.

```
In[81]:= Export["Dataset_json.json",user,"JSON"]
Out[81]= Dataset_json.json
```

If you have a dataset of repeated keys, you can export it to the JSON format (see Figure 4-17).

```
In[82]:= assoc1=<|"Log in Date"->DateObject[{2020,06,29}], "User ID"->
123, "Status"->"Active"|>;
assoc2=<|"Log in Date"->DateObject[{2020,06,28}], "User ID"->122, "Status"->
"Not Active"|>;Dataset[{assoc1,assoc2}]
Export["Dataset2_json.json",%, "JSON"]
Out[83]=
```

Log in Date	User ID	Status
Mon 29 Jun 2020	123	Active
Sun 28 Jun 2020	122	Not Active

**Figure 4-17.** *User Dataset*

```
Out[84]= Dataset2_json.json
```

To be precise, you can export shapes where the dataset contains complex structures, such as an association of associations. Let’s look at the following example, which builds a dataset (see Figure 4-18).

```
In[85]:= assoc3="Player A"->Association["Date"->DateObject[{2020,06,29}],
"User ID"->123,"Status"->"Active"];assoc4="Player B"->Association["Date"->
DateObject[{2020,06,28}], "User ID"->122,"Status"->
"Not Active"];Dataset[{<|assoc3,assoc4|>}]
Out[85]=
```

	Date	User ID	Status
Player A	Mon 29 Jun 2020	123	Active
Player B	Sun 28 Jun 2020	122	Not Active

**Figure 4-18.** Tagged dataset

Subsequently, proceed to export the dataset.

```
In[86]:= Export["Dataset3_json.json",%,"JSON"]
Out[86]= Dataset3_json.json
```

Let’s try to better understand how to export in JSON format. When you export information such as a rule list or a single association, the structure of the content in the exported JSON file is through a collection of pairs between braces and values. On the contrary, when you have ordered structures, such as an association of lists and an association of associations, the structure of the content in the JSON file is as an ordered array within the array of the collections of associated pairs between braces and values. Quite the opposite; however, exporting a nested list is already in the form of sorted arrays. To clarify this, the reader can observe how a list of rules is exported through the following code.

```
In[87]:= rules={"apple"->3,"car"->"3","2"->2};
Export["Rules.json",rules,"JSON"]
Out[88]= Rules.json
```

In addition, for a nested list or list of lists.

```
In[89]:= array=Array[{#1,#2}&,{4,4}]
Export["Array.json",array,"JSON"]
Out[89]= {{{{1,1},{1,2},{1,3},{1,4}},{2,1},{2,2},{2,3},{2,4}},{3,1},{3,2},
{3,3},{3,4}},{4,1},{4,2},{4,3},{4,4}}}}
Out[90]= Array.json
```

If the created file is observed, it must contain an array of arrays inside the JSON file.

## Content File Objects

It should be concluded that for all the exported files, you can create a content object showing you the properties of the created files. This is done with the `ContentObject` function, which provides content from a file. Let's use the association's example to create a JSON file to do this.

```
In[91]:= Association@{"Name"->"Ellis","Date of birth"->DateObject[{1990,01,04]},
"Height"->Quantity[180,"Centimeters"],"Favorite color"->"Red","Hobbies"->
"Soccer, Pc gaming, Board games","Social networks"->"Twitter, Facebook"};
user=Dataset[%];
jsonFile=Export["Dataset_json_2.json",user,"JSON"];
```

Now, you need to get the path where the file is located with `AbsoluteFileName`.

```
In[94]:= AbsoluteFileName[jsonFile]
Out[94]= /Users/macosex/Desktop/Dataset_json_2.json
```

Let's now use the file to create the file object type representation. Then, `ContentObject` is applied to the file object.

```
In[95]:= ContentObject[%]
Out[95]=
```

A content-type object appears (see Figure 4-19).



```
ContentObject[ Plaintext: /Users/macosex/Desktop/Dataset_json_2.json
CreationDate: Tue 21 Nov 2023 20:56:53]
```

**Figure 4-19.** *ContentObject for the JSON files created*

Pressing the + icon provides you with the exported file's properties, such as name, size, creation dates, and file localization. You can access the properties programmatically using the following form.

```
In[96]:= ContentObject[%]["Properties"]
Out[96]= {CreationDate,Plaintext}
```

This can be applied to other exported files.

## Searching Files with Wolfram Language

With the Wolfram Language, you can look at the location of the file or files.

The NotebookDirectory command is used to see the path of the notebook directory. It shows the full directory containing the notebook in which you work.

```
In[97]:= NotebookDirectory[]
Out[97]= /Users/macosex/Desktop/
```

Now, SetDirectory is used to set a working directory as the current directory. You can enter the path of the desired directory and establish it as the working directory. However, now set the notebook directory as the new working directory.

```
In[98]:= SetDirectory[NotebookDirectory[]]
Out[98]= /Users/macosex/Desktop
```

With this new directory set, you can locate files in the new directory, the notebook location. Here, the FileNames command lets you explore files in the working directory, which, in this case, is the notebook's directory because it was set up in the previous code.

```
In[99]:= FileNames[]
Out[99]= {Color_table.txt,Grocery_List.csv,Hello_World,Hello_World.
txt,import export.nb,weather.csv}
```

FileNames show all types of files available in the directory. If you have many files in the directory, you can search for a particular file by using FindFile and entering the file's name as a string. The full path of the file is displayed.

```
In[100]:= FindFile["Color_table.txt"]
Out[100]= /Users/macosex/Desktop/Color_Table.txt
```



File extensions can be searched, too.

```
In[101]:= FileNames["*.txt"]
Out[101]= {Color_table.txt,File_text.txt,Hello_World.txt,New_File.txt}
```

---

**Note** Other types of File commands exist; to look for more commands associated with the name file, enter `??File*`.

---

Remember, this is when you set the working directory as the notebook directory. If you have not set a directory previously, Mathematica searches the default directories of your machine, which are the ones shown entering `$Path`.

## Connecting to External Services

Besides export and import capabilities, Mathematica can connect to various external services, like external resources, external connectivity, and database management through external evaluations.

### External Connections

With the launch of Mathematica version 13, improvements have been put together, especially in connecting with external services. One notable feature is external evaluators, which enable interaction with various languages such as Julia, Ruby, R, Python, Java, Octave, Node.js, Shell, and SQL. To discover and utilize installed evaluator systems, use `FindExternalEvaluators`, which scans standard directories for use in any local evaluation.

Executing `FindExternalEvaluators[]`, with no arguments, searches for all available languages installed on your computer. Let's find the version of the Shell evaluator. On macOS, it usually refers to the Bash shell; on Windows, it's typically PowerShell.

```
In[102]:= FindExternalEvaluators["Shell"]//Normal//Print
Out[102] =
<|4ce695dd-ef6a-7006-f30d-b4320329bbd7 → <|System → Shell,
  Version → 3.2.57, Target := /bin/bash, Executable := /bin/bash,
  Registered → Automatic|>,
```

```

b217afb1-d97f-3cfa-3c52-19ec78df64bc → <|System → Shell,
  Version → 3.2.57, Target := /bin/sh, Executable := /bin/sh,
Registered → Automatic|>,
342330ff-7009-e5ec-c00a-86949f3c0f7a → <|System → Shell,
  Version → 5.9, Target := /bin/zsh, Executable := /bin/zsh,
Registered → Automatic|>|>

```

In this case, the output lists three shell versions: Bash (Bourne Again SHell), Sh (Bourne Shell), and Zsh (Z Shell).

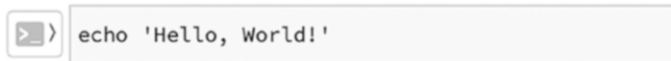
---

**Note** The external language cannot be used if the Registered value is not set to True or Automatic. For troubleshooting, go to the Wolfram documentation page at <https://reference.wolfram.com/language/workflowguide/ConnectingToExternalSoftware.html>.

---

Once the external evaluator has been registered, it can be used with ExternalEvaluator. You can use ExternalEvaluate by applying the function directly or, in a new cell, by typing ‘>’ to initiate a command line, where a yellow block line appears. Choose your language from a drop-down list on the left icon or input it directly as a string and then the code, as shown in Figure 4-20.

```
ExternalEvaluate["Shell", "echo 'Hello World!'"]
```



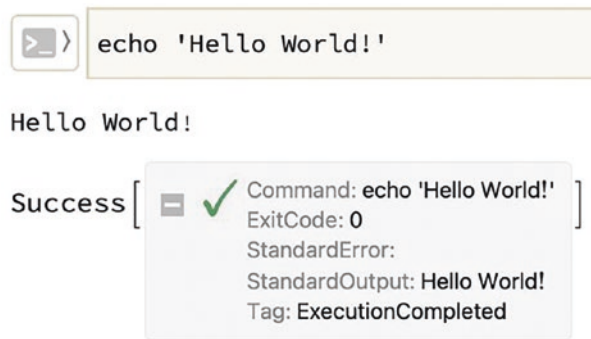
**Figure 4-20.** External evaluation for Z shell code using the ExternalEvaluate and the ‘>’ type command block

Executing the code following prints “Hello World!” using the Z shell. The resulting exit code is 0, signifying success, and is displayed as standard notebook output (see Figure 4-21).

```

In[103]:=
Out[103]=

```



**Figure 4-21.** External evaluation using Z shell code Hello World!

Different prerequisites may be required, such as additional libraries and the language executable, depending on the external language you intend to use. While language cells are handy, ExternalEvaluate offers more programmatic output flexibility.

## External Resources

The prior section highlights ExternalEvaluate’s role in integrating outer languages in a notebook. Despite this, Mathematica can generate and utilize outer resources like outer functions. Node.js version 21.2.0 was used while creating this book. It can be installed from the official site or approved repositories. In this case, the Homebrew package installer was used. Using Node.js required the zeromq library, installed using npm, as stated in the Wolfram documentation.

---

**Note** For detailed info, visit Wolfram documentation. NodeJS for ExternalEvaluate: <https://reference.wolfram.com/language/workflow/ConfigureNodeJSForExternalEvaluate.html>

---

To automatically identify Node.js, use FindExternalEvaluators[“NodeJS”], similar to the shell language process. If successful, Registered shows as Automatic, indicating complete setup. If MissingDependencies appears, Mathematica can’t find the necessary dependencies, requiring manual registration. Regardless, it’s advised to manually register the external evaluator by adding the executable’s path to ensure proper function.

Like in the shell process, to autodetect Node.js, use `FindExternalEvaluators["NodeJS"]`. If `Registered` shows as `Automatic`, all setup is done. If `MissingDependencies` shows, Mathematica lacks needed dependencies, requiring manual registration. Regardless, you should manually register the external evaluator by adding the executable's path to ensure proper function.

```
In[104]:= RegisterExternalEvaluator["NodeJS", "/opt/homebrew/bin/node"]
Out[104]= 629ba62a-8d17-e9fe-6cd9-870f94c7933c
```

Then, trying to find it again.

```
In[105]:= FindExternalEvaluators["NodeJS"] // Normal // Print
Out[105]=
<|629ba62a-8d17-e9fe-6cd9-870f94c7933c → <|System → NodeJS,
  Version → 21.2.0, Target ⇒ /opt/homebrew/bin/node,
  Executable ⇒ /opt/homebrew/bin/node, Registered → True|>|>
```

The `Registered` key has a value of `True`, meaning successful manual registration. To test it, calculate the square root of 25.

```
In[106]:= ExternalEvaluate["NodeJS", "Math.sqrt(25)"]
Out[106]= 5
```

With Node.js set, custom functions can be implemented—for instance, a primary function to find the square root of a number.

```
In[107]:= jsFun1 =ExternalFunction["NodeJS", "Math.sqrt"]
Out[107]= ExternalFunction[System : NodeJS Command : Math. sqrt
Session : Automatic ]
```

The outer Node.js system calculates using `Math.sqrt`. If no external session is manually set, it is automatic. The function is now at hand in the notebook.

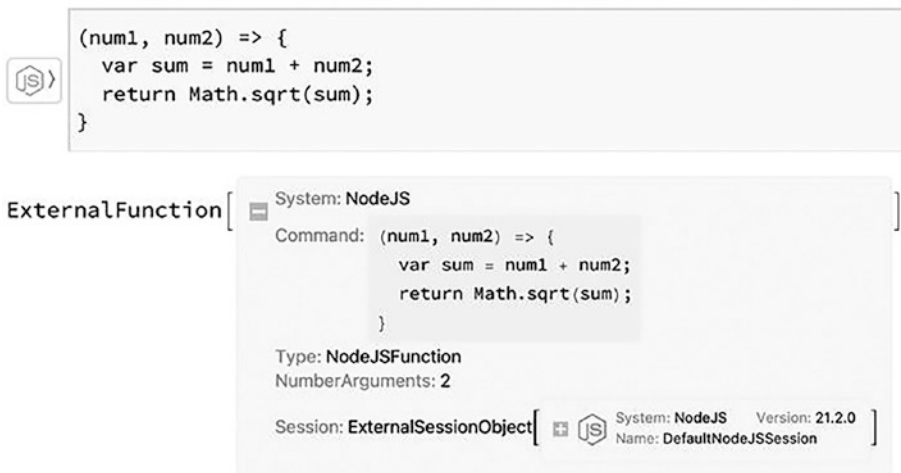
```
In[108]:= jsFun1[#]&@{25,36,49,64}
Out[108]= {5,6,7,8}
```

The `Function` syntax can vary, but the process is the same; for example, using an arrow function.

```
In[109]:= jsFun2 =
ExternalFunction["NodeJS", "(number) => Math.sqrt(number);"];
jsFun2[#] &/@ {25,36,49,64}
Out[109]= {5,6,7,8}
```

Using the external block is also at hand. Figure 4-22 shows that the node.js function is linked to a default external node.js session.

```
In[111]:=
Out[111]=
```



**Figure 4-22.** Node.js function to return the square root of the sum of two numbers

```
In[113]:= %[18,18]
Out[113]= 6
```

---

**Note** To ensure that a function can be called in NodeJS using ExternalFunction, it must be explicitly returned.

---

To unregister an external evaluator, type the system language and the executable path. In this case, it is the same path used when registered.

```
In[114]:= UnregisterExternalEvaluator["NodeJS", "/opt/homebrew/bin/node"]
Out[114]= 629ba62a-8d17-e9fe-6cd9-870f94c7933c
```

## Database and File Operations (SQL)

Database and file operations can be performed in Mathematica using external languages like SQL. By leveraging `ExternalEvaluate`, it is possible to execute SQL queries and work directly with dataset formats.

You can generate a reference object for the database by utilizing a table from the example data folder.

```
In[115]:= DatabaseReference[FindFile["ExampleData/ecommerce-database.
sqlite"]];
Shallow[%]
Out[116]//Shallow=
DatabaseReference[<|Backend → SQLite, Name → /Applications/Mathematica.
app/Contents/Documentation/English/System/ExampleData/ecommerce-database.
sqlite|>]
```

The reference of the retrieved file with `FindFile` associates the `.sqlite` local file with the backend SQL engine set as `SQLite`, which performs operations and data management.


---

**Note** SQL should be available within Mathematica, but check that it appears as a registered external evaluator `FindExternalEvaluator["SQL"]`. If not, make sure to register the evaluator.

---

After referencing, view all database table names (see Figure 4-23). Choose a table (offices) (see Figure 4-24) and select territory and city, ordering by territory (see Figure 4-25).


```
In[117]:=
Out[117]=
```

	SELECT name FROM sqlite_master WHERE type='table'
name	
offices	
productlines	
employees	
products	
customers	
orders	
payments	
orderdetails	

**Figure 4-23.** Listing all tables

In[118]:=

Out[118]=



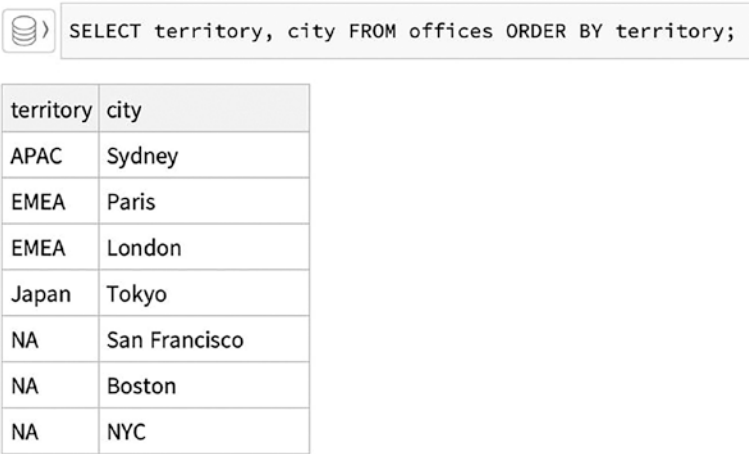
SELECT \* FROM offices;

officeCode	city	phone	addressLine1	addressLine2	state	country	postalCode	territory
1	San Francisco	+1 650 219 4782	100 Market Street	Suite 300	CA	USA	94080	NA
2	Boston	+1 215 837 0825	1550 Court Place	Suite 102	MA	USA	02107	NA
3	NYC	+1 212 555 3000	523 East 53rd Street	apt. 5A	NY	USA	10022	NA
4	Paris	+33 14 723 4404	43 Rue Jouffroy D'abbans	—	—	France	75017	EMEA
5	Tokyo	+81 33 224 5000	4-1 Kioicho	—	Chiyoda-Ku	Japan	102-8578	Japan
6	Sydney	+61 2 9264 2451	5-11 Wentworth Avenue	Floor #12	—	Australia	NSW 2010	APAC
7	London	+44 20 7877 2041	25 Old Broad Street	Level 7	—	UK	EC2N 1HN	EMEA

**Figure 4-24.** Fetching all office data

In[119]:=

Out[119]=



**Figure 4-25.** *Sorting offices by territory*

# Summary

This chapter explored essential aspects of importing and exporting various file formats, including costume imports. It provides the basics of semantic import, dealing with quantities and large datasets. The chapter also offered a deep dive into data management and the search of content file objects within a notebook. The chapter concluded with a discussion on connecting to external elements, establishing connections, and working with external resources, databases, and files.



## CHAPTER 5

# Data Visualization

This chapter discusses data visualization in more depth, showing the different ways of visually representing data, using different commands, and creating a range of different types of graphs. It also explains how to customize plots and use predefined plot themes.

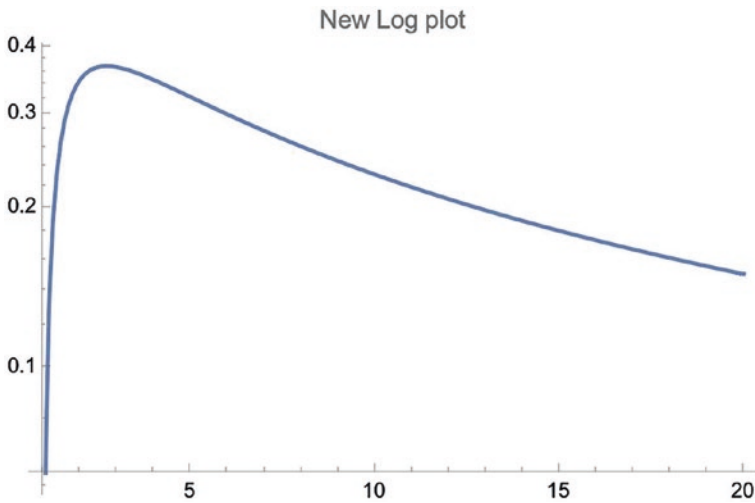
## Basic Visualization

Data visualization is key for understanding information about data. Visual tools such as 2D plots, contour plots, 3D plots, and time series provide a handy form to view and understand trends and patterns of the data. One of the things about Wolfram Language is that it contains commands that enable you to plot graphs in a simple form. Now, you can better learn how plotting works. Mathematica treats every plot as a graphic object, that is because every graphic is created of primitive elements (points, lines, polygons, geometric figures, etc.), directives (style, shape, size, width, blurriness, etc.), and options (visual modifications, styles, frames, aspects, text, etc.). However, let's focus on the area of 2D and 3D plots.

## 2D Plots

Simple 2D plots over a specified range are relatively simple to create, as you saw in Chapter 1 with the `Plot` function. The Wolfram Language gives you accurate control over your plots; for example, you can define the range of your plot's range and many options. For instance, you can add a title to the next plot, a `LogPlot`, which is a function in a logarithm scale (see Figure 5-1).

```
In[1]:= LogPlot[Log[x]/x,{x,1,20},PlotLabel->"New Log plot"]
Out[1]=
```

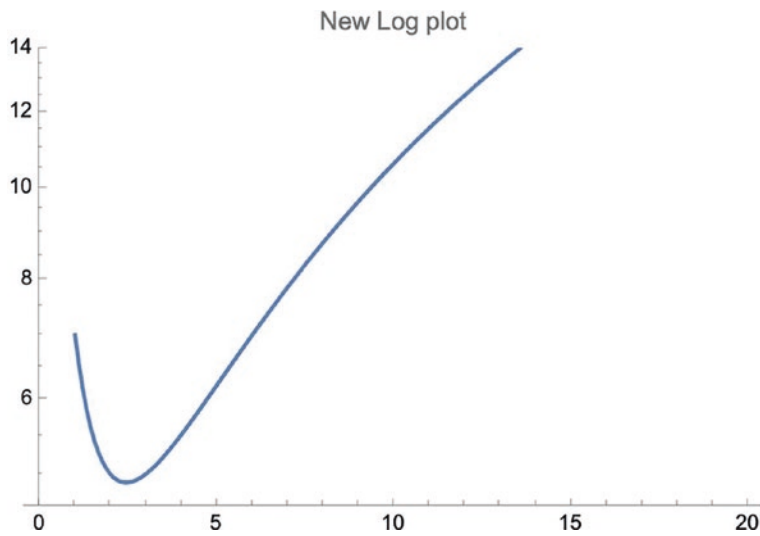


**Figure 5-1.** *LogPlot*

Figure 5-1 shows that a title has been added.

When plotting points over an interval, the default plot range to show is produced automatically by Mathematica. But with `PlotRange`, you can override the option and enter a desired range (see Figure 5-2).

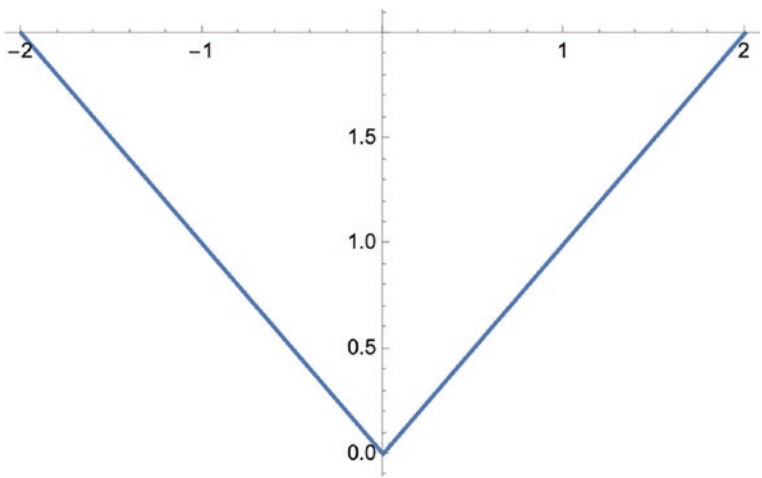
```
In[2]:= LogPlot[x+(6/x),{x,1,20},PlotLabel->"New Log
plot",PlotRange->{0,14}]
Out[2]=
```



**Figure 5-2.** *LogPlot of  $x+(6/x)$ , with custom range*

By selecting All in PlotRange, the y axis increases. Alternatively, you can choose the limits by entering them in the form {y min, y max}. Sometimes, a graphic may not pass through a desired set of coordinates; to force this, AxesOrigin is used (see Figure 5-3). Intersections are written in the form {x,y}, where the coordinates denote the x and y origin points.

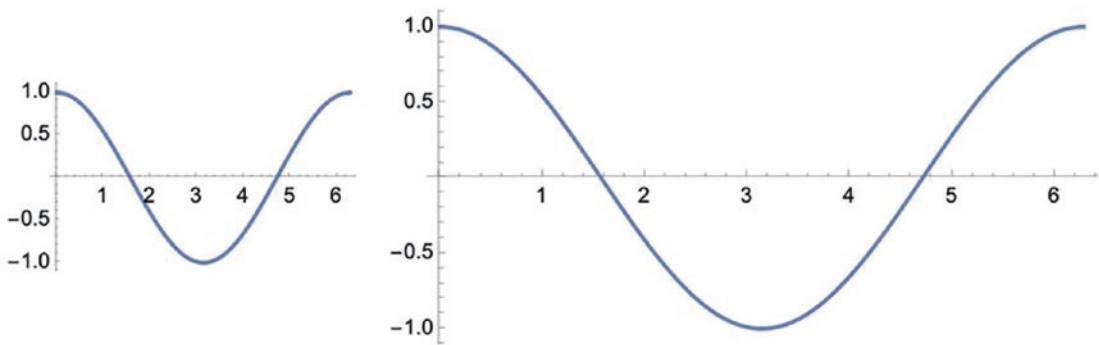
```
In[3]:= Plot[Abs[x],{x,-2,2}, AxesOrigin->{0,2}]
Out[3]=
```



**Figure 5-3.** *The absolute value of  $x$  on origin 0, 2*

`AspectRatio` is used to control the aspects using their height and width. This option allows you to specify how big or small a graphic can be, calculating the height and width ratio (h/w). However, when using `ImageSize` to directly select the width and height of a graphic, if you specify the height alone, it is better to set `AspectRatio` to `Full`. This ensures proper scaling as the width adjusts accordingly. Both options are shown in Figure 5-4.

```
In[4]:=GraphicsRow[{Plot[Cos[x],{x,0,2\[Pi]},ImageSize->Small],
Plot[Cos[x],{x,0,2\[Pi]},AspectRatio->0.5]}]
Out[4]=
```

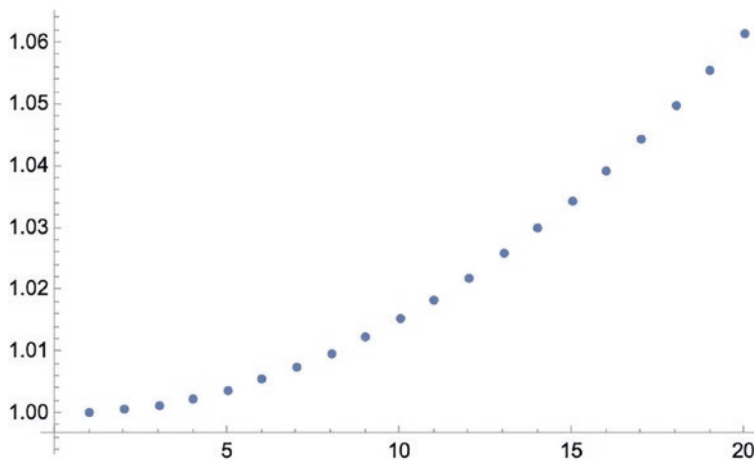


**Figure 5-4.** *First graphic with `ImageSize`; second with `AspectRatio`*

## Plotting Data

When plotting graphs, a set of points can be represented in a plot. Data can be plotted with different commands, depending on their purpose. To plot a list of coordinates, `ListPlot` is used, and the arguments of the plot are represented as x, y coordinates (`{x1,y1}, {x2,y2} ...`). You can create a list of values and pass them as the arguments. The following example creates a table of values to resemble a hyperbolic cosine, with one step between each point (see Figure 5-5).

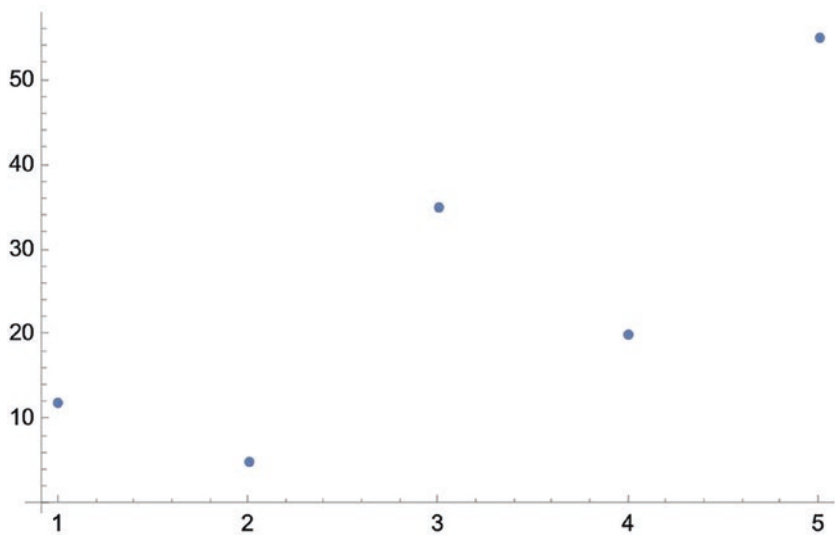
```
In[5]:= ListPlot[Table[Cosh[i Degree],{i,1,20}]]
Out[5]=
```



**Figure 5-5.** Hyperbolic cosine plot, ranging from 1 to 20

In this case, you only generate points in `{1, y1}, {2, y2}`, but you can also plot x and y values. Let's generate the x points with `Table` and then thread each element of x to a y element and plot (see Figure 5-6) the new set of coordinates.

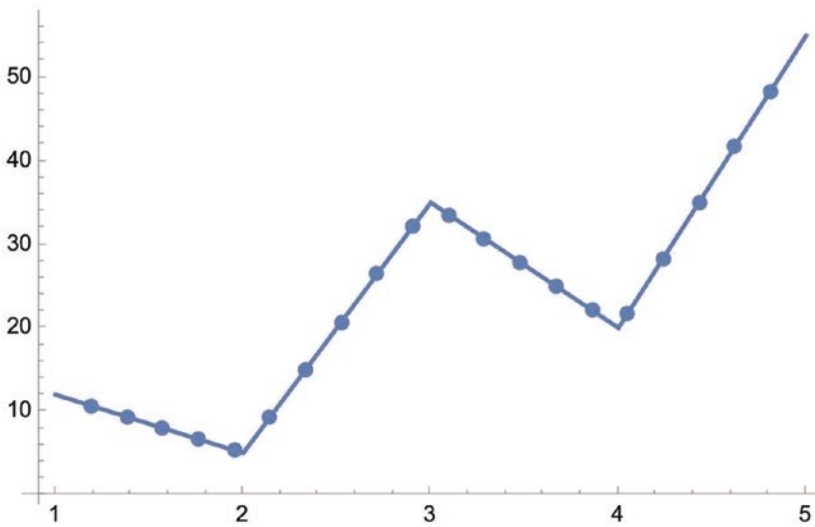
```
In[6]:= xcoor=Table[i,{i,1,5}];
ycoor={12,5,35,20,55};
coordinates=Thread[{xcoor,ycoor}];
ListPlot[coordinates]
Out[9]=
```



**Figure 5-6.** *ListPlot of  $x$  and  $y$  coordinates*

Another useful command is ListLinePlot, which plots points through points by joining them with a line. ListLinePlot (see Figure 5-7) can also plot predefined coordinates. You can show how many points to display to understand how the plot is constructed with the Mesh option.

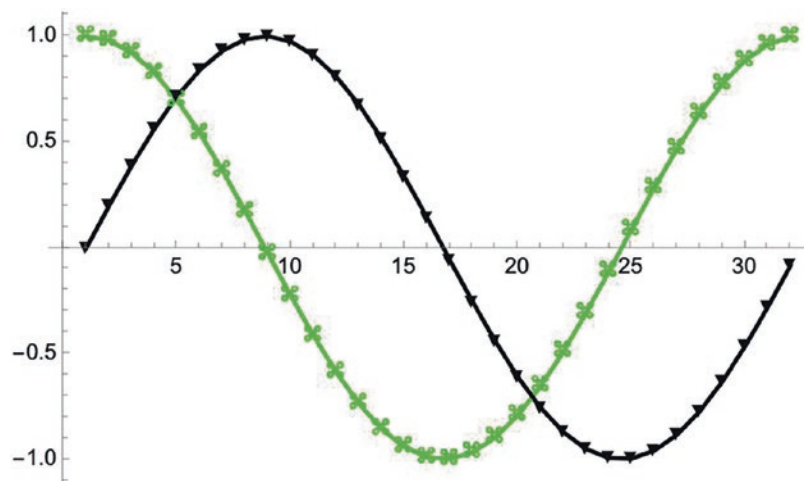
```
In[10]:= ListLinePlot[coordinates,Mesh->20]
Out[10]=
```



**Figure 5-7.** *ListLinePlot with mesh option set to 20*

A plot can be represented with different colors and markers. Colors and markers are convenient to distinguish among different plots. To introduce markers, enter the `PlotMarkers` option followed by the markers symbol. Markers can be special characters or letters; use the special character pallet for a complete list of symbols and characters. By default, different sets are colored differently, but to choose a specific color, use `PlotStyle`. With `PlotStyle` the thickness of a line can be changed too, as shown in Figure 5-8.

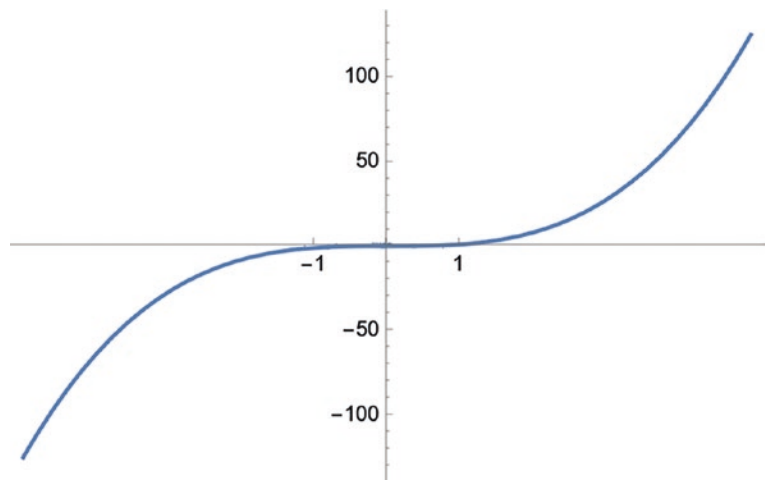
```
In[11]:=
ListLinePlot[{Table[Cos[i], {i, 0, 2 \[Pi], 0.2}],
  Table[Sin[i], {i, 0, 2 \[Pi], 0.2}]], PlotMarkers -> {"\[CloverLeaf]", "\[FilledDownTriangle]"}, PlotStyle -> {Green, Black}]
Out[11]=
```



**Figure 5-8.** *Plots with different marker points*

Another general option is `Ticks`. With this option, you can modify the indicators on the axes for both `x` and `y`. For example, in Figure 5-9, the plot ticks are marked on the `x` axis; the ticks are `-1` and `1`. And the `y` axis is set to automatic (see Figure 5-9).

```
In[12]:= Plot[x^3,{x,-5,5},Ticks->{{-1,0,1},Automatic}]
Out[12]=
```

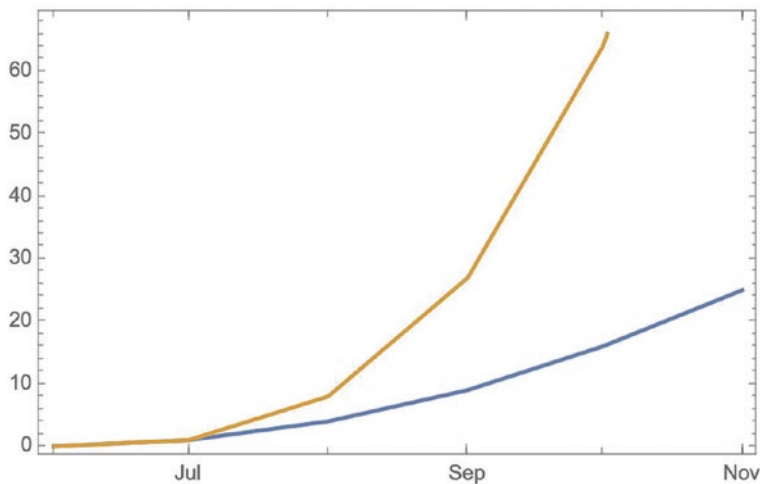


**Figure 5-9.** *Plots with ticks marked on -1 and 1 for the x axis*



Additionally, plots containing dates can be displayed with `DateListPlot`. The `DateListPlot` has the following form, `DateListPlot[{v1,v2, ... }, "date specification"]`. With `DateListPlot`, the x axis is converted into a timeline, and the y axis corresponds to the values (v1,v2, ...). Figure 5-10 shows a `DateListPlot`, starting in June and finishing in November.

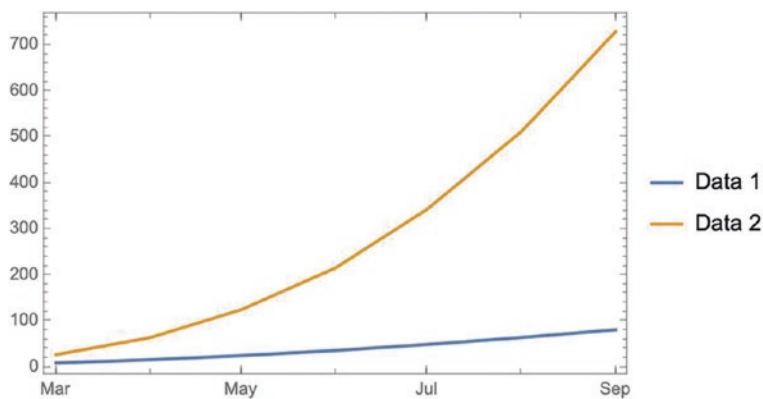
```
In[13]:= data1=Table[Power[i,2],{i,0,5}];
data2=Table[Power[i,3],{i,0,5}];
DateListPlot[{data1,data2},{2006,06}]
Out[15]=
```



**Figure 5-10.** Date plot, starting the plot from June 2006 to November 2006

Additionally, you can use `ListLinePlot` or `ListPlot` to create date plots. Employing the `ScalingFunction` option with `{"Date", Identity}` allows a proper scaling along the date axis, for good data visualization over time, as the following code and Figure 5-11 show.

```
In[16]:= data1=Table[{DateObject[{2006,i}],Power[i,2]},{i,3,9}];
data2=Table[{DateObject[{2006,i}],Power[i,3]},{i,3,9}];
ListLinePlot[{data1,data2},ScalingFunctions->{"Date",Identity},PlotStyle->
Automatic,Frame->True,PlotLegends->{"Data 1","Data 2"}]
Out[17]=
```

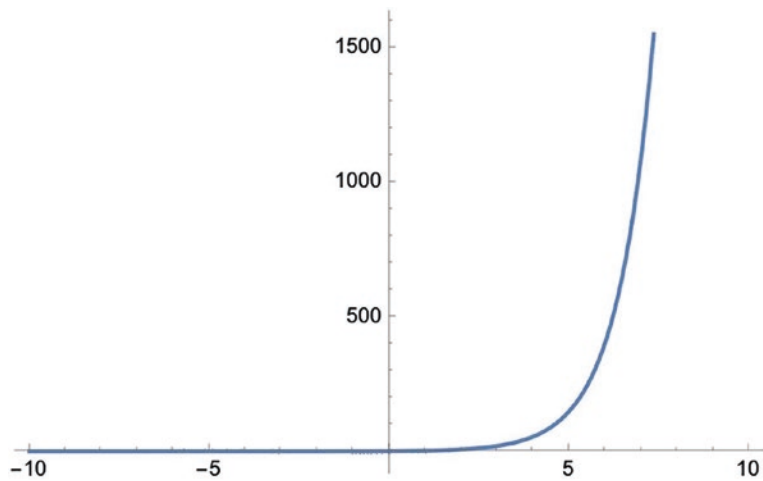


**Figure 5-11.** Date plot using *ListLinePlot* with *ScalingFunctions*

## Plotting Defined Functions

You can define and plot custom functions (see Figure 5-12). User functions can also be used as arguments for plotting commands. Functions can have a single or multiple variables, as with 3D plots.

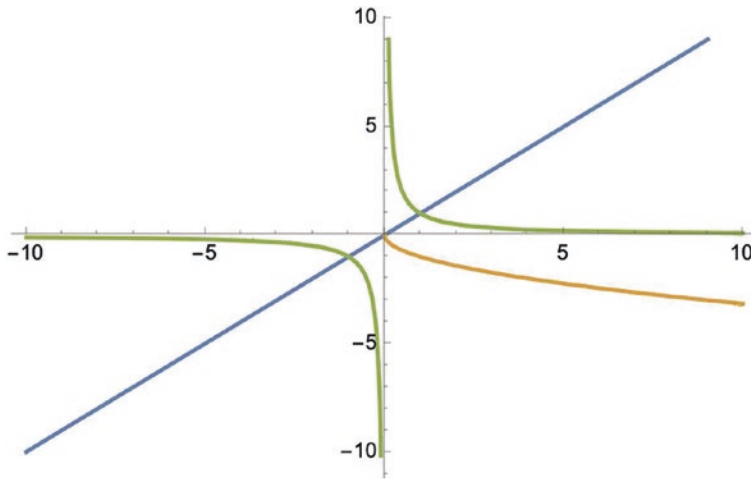
```
In[17]:= F[x_]:=Exp[x];  
Plot[F[x],{x,-10,10}]  
Out[18]=
```



**Figure 5-12.** User-defined function for *Exp* of *x*

Also, multiple defined functions are supported. When multiple plots are in the same graphic, each plot is colored differently (see Figure 5-13).

```
In[18]:= X[x_]:=x;Y[y_]:=-Sqrt[y];Z[z_]:=1/z;
Plot[{X[x],Y[x],Z[x]},{x,-10,10}]
Out[19]=
```



**Figure 5-13.** Multiple plots

## Customizing Plots

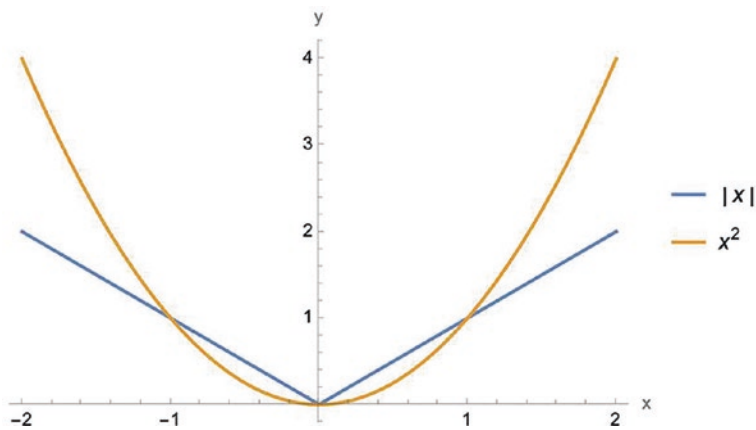
The Wolfram Language lets users customize plots based on their needs, like adding text, changing color style, adding fill, presenting on tabular frameworks, and so forth. Many commands used in the 2D plots are also preserved in 3D plots. Depending on the graphical representation, options can vary between commands.

## Adding Text to Charts

Adding text to charts, like markers and the range of values, can make a chart more informative. Many other elements can be added too.

PlotLabel adds a title to a chart. In addition to this option, there is AxesLabel and PlotLegends. The first allows you to add labels to your axes in the form {"x\_label," "y\_label"}; the second enables you to add text related to each expression within the graph (see Figure 5-14).

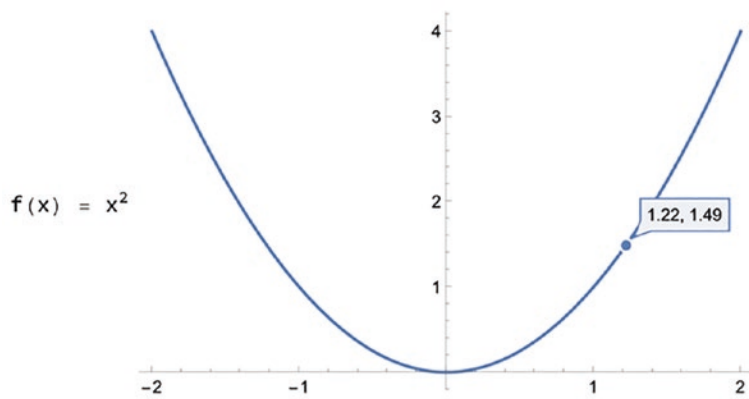
```
In[20]:= Plot[{Abs[x], x^2}, {x, -2, 2}, AxesLabel -> {"x", "y"},
  PlotLegends -> "Expressions"]
Out[20]=
```



**Figure 5-14.** *Plots with labeled axes and functions*

You can use Labeled to add costume text expressions on plots (see Figure 5-15). As for the new Mathematica version, passing the cursor over the plot displays the x and y coordinates without creating an explicit tooltip.

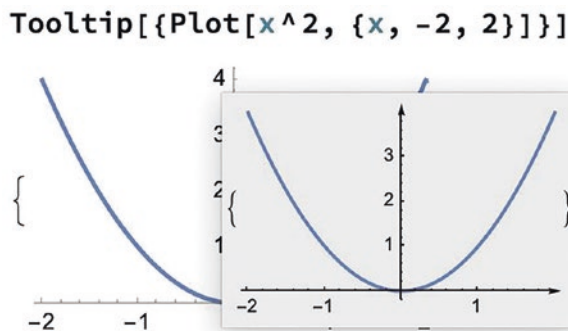
```
In[21]:= Labeled[Plot[x^2, {x,-2,2}], "f(x) = x^2,, Left]
Out[21]=
```



**Figure 5-15.** *Label placed on the left side of the graphic*

Even with the `Labeled` command, Tooltips can be constructed. Tooltips display a label tooltip for any expression (see Figure 5-16). Tooltips are displayed when the mouse pointer is passed over the tooltip expression. The difference between Tooltips and PlotLegends is that PlotLegends is an option and not a command.

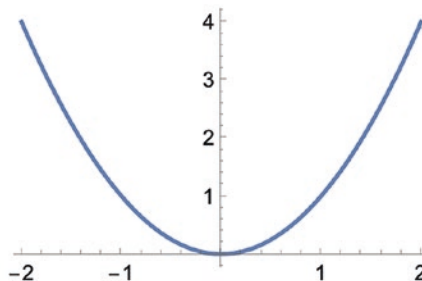
```
In[22]:= Tooltip[Plot[x^2,{x,-2,2}]]
Out[22]= {}
```



**Figure 5-16.** Tooltip created for the plot expression

When you hover over the entire graph, it shows you the tooltip of the entire graph since you specify it. But you can do it just for the expression of the function (see Figure 5-17).

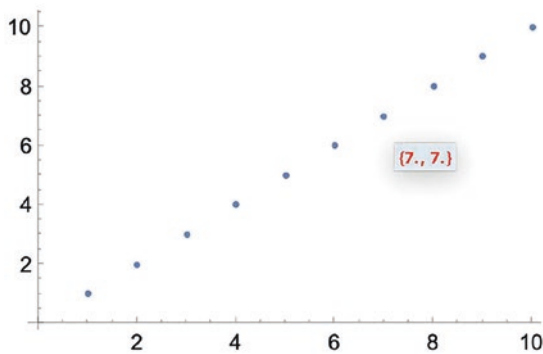
```
In[23]:= Plot[Tooltip[x^2],{x,-2,2},ImageSize->200]
Out[23]=
```



**Figure 5-17.** Tooltip for the curve expression

If you hover over the curve, it shows you the tooltip of  $x^2$ ; this function also works with the other types of plots. You can add what the tooltip style should look like with the `ToolTipStyle` option (see Figure 5-18).

```
In[24]:=ListPlot[Tooltip[Range[10], TooltipStyle -> {Bold,Red,
Background -> LightBlue}], ImageSize -> 250]
Out[24]=
```



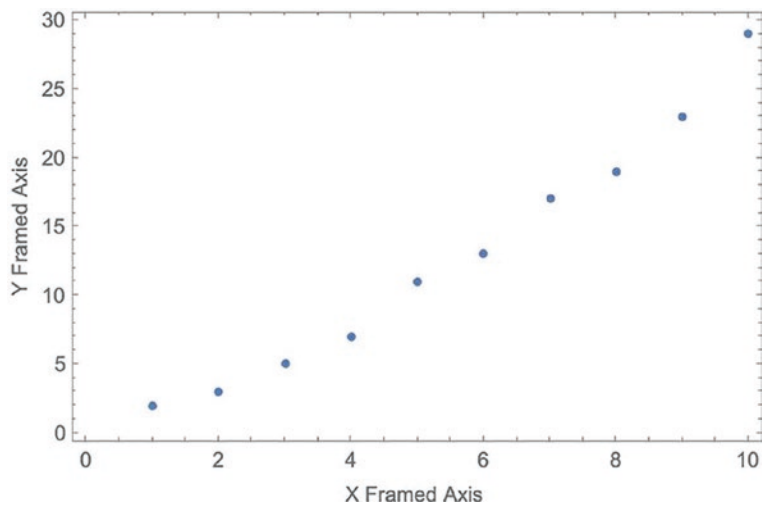
**Figure 5-18.** *Tooltip for every point plotted*

If you move the cursor to the points, you get the coordinates of the points written in red and the tooltip's background in light blue.

## Frame and Grids

Plots can be framed and gridded. The `Frame` option is used, and to add labels to the frame, use `FrameLabel`, which receives instructions like `AxesLabel` (see Figure 5-19).

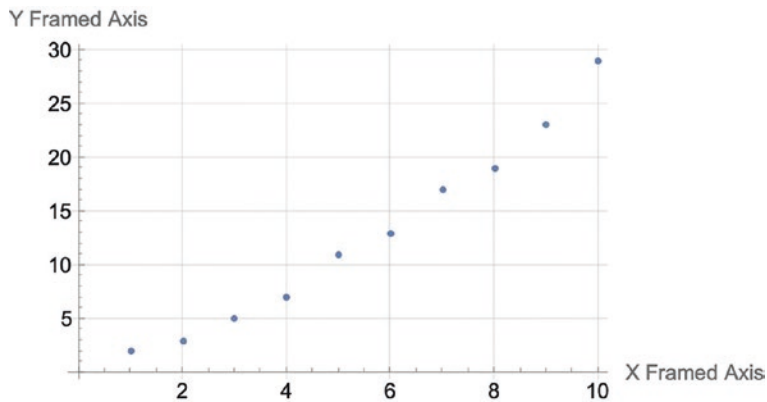
```
In[25]:= ListPlot[Table[Prime[i],{i,1,10}],Frame->True,FrameLabel->
{"X Framed Axis ","Y Framed Axis"}]
Out[25]=
```



**Figure 5-19.** *Framed ListPlot*

To add a grid (see Figure 5-20), use the `GridLines` option.

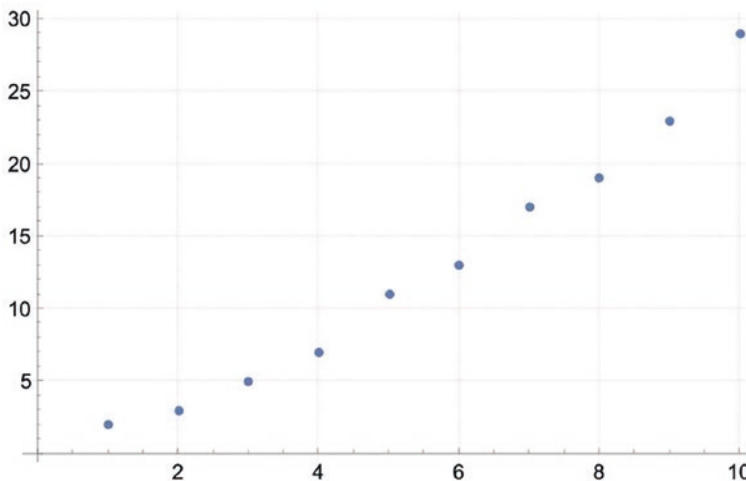
```
In[26]:= ListPlot[Table[Prime[i],{i,1,10}],GridLines->Automatic,AxesLabel->
{"X Framed Axis ","Y Framed Axis"}]
Out[26]=
```



**Figure 5-20.** *Gridded plot*

To modify the grid style, use the `GridLinesStyle` option, which can have a particular thickness using `Directive` (see Figure 5-21).

```
In[27]:= ListPlot[Table[Prime[i], {i, 1, 10}], GridLines -> Automatic,
GridLinesStyle -> Directive[Thickness[0.0002], LightRed]]
Out[27]=
```



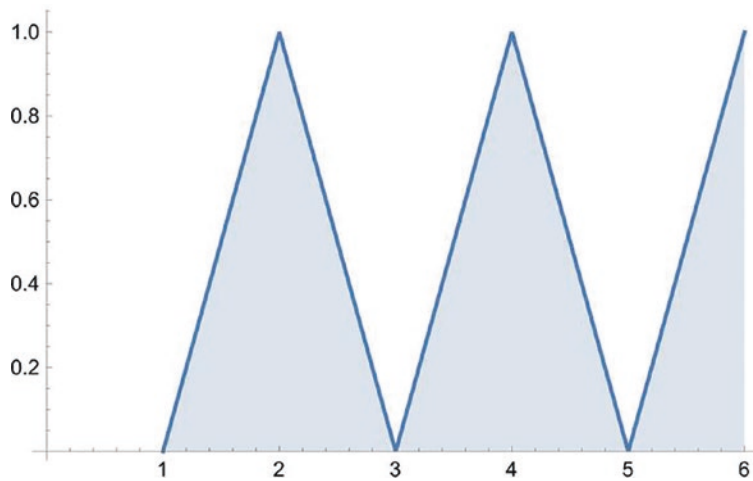
**Figure 5-21.** *GridLines colored in light red*

## Filled Plots

Plots can be filled in various forms—for example, between the x axis, from the bottom and top of a curve (see Figure 5-22).

```
In[28]:= ListLinePlot[Table[Mod[i,2],{i,0,5}],Filling->Bottom]
Out[28]=
```

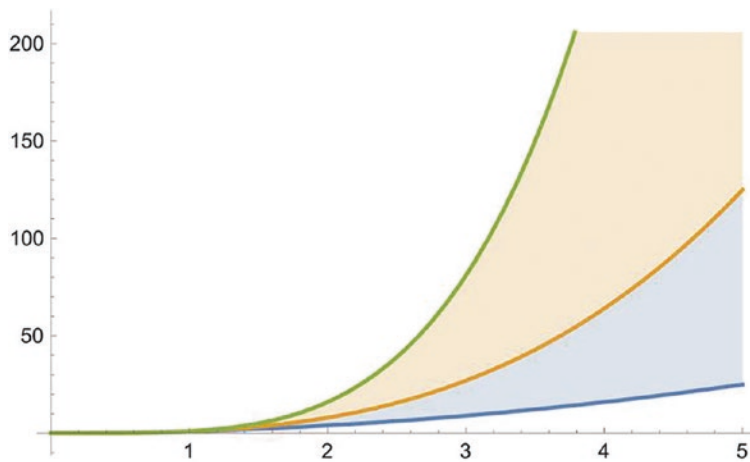




**Figure 5-22.** Filled plot from plotted points to the bottom of the axis

A specified region between curves can also fill them by introducing `Filling` → {"1st curve" → {"2nd curve"}, "2nd curve" → {"3rd curve"}}, as shown in Figure 5-23.

```
In[29]:= Plot[{x^2, x^3, x^4}, {x, 0, 5}, Filling -> {1 -> {2}, 2 -> {3}}]
Out[29]=
```

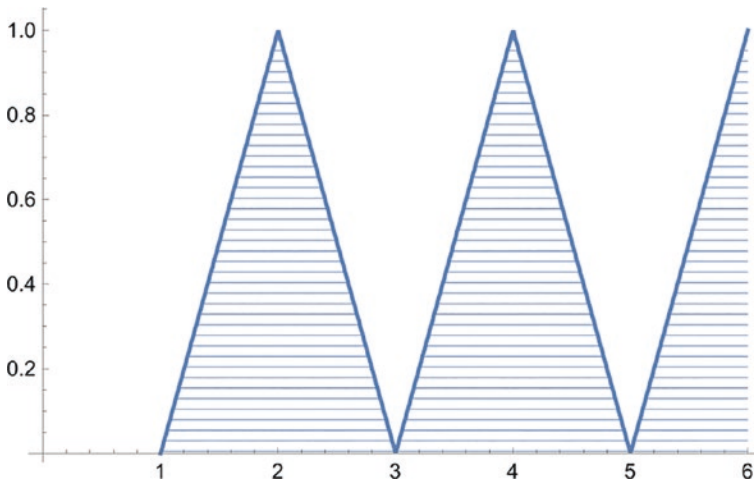


**Figure 5-23.** Filled plots

## Filling Patterns and Gradient

The updated version has added new features, such as cross-hatching fillers. This enhancement is used like the standard options illustrated in Figure 5-24.

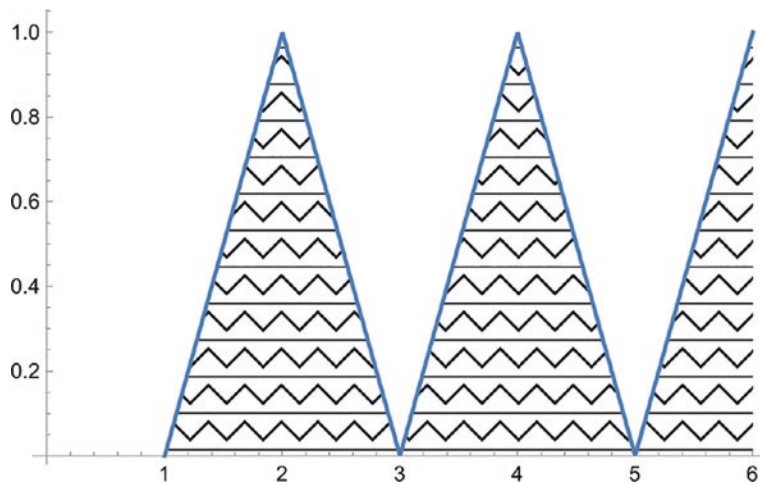
```
In[30]:= ListLinePlot[Table[Mod[i,2], {i,0,5}], Filling -> Bottom,
FillingStyle -> HatchFilling["Horizontal"]]
Out[30]=
```



**Figure 5-24.** Filled horizontal style

New function additions are implemented by a style or a pattern, as seen in Figure 5-25.

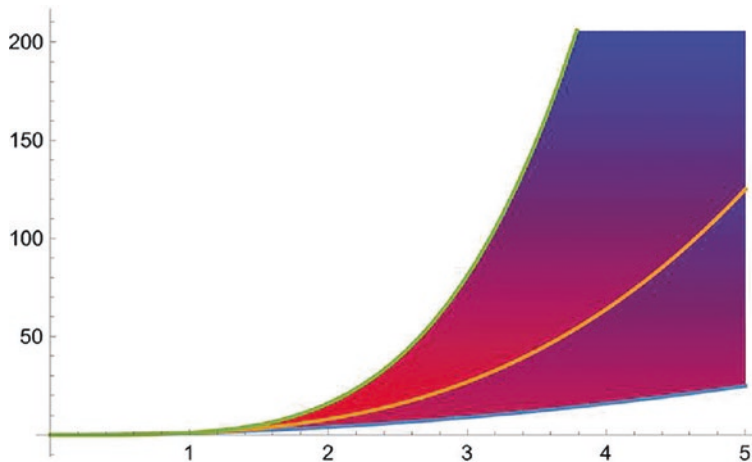
```
In[31]:= ListLinePlot[Table[Mod[i,2], {i, 0, 5}], Filling -> Bottom,
FillingStyle -> PatternFilling["ChevronLine", ImageScaled[1/20]]]
Out[31]=
```



**Figure 5-25.** Filled Chevron horizontal line style

The same applies to shading functions; additions are implemented by the gradient technique, as seen in Figure 5-26.

```
In[32]:= Plot[{x^2,x^3,x^4}, {x,0,5},FillingStyle -> LinearGradientFilling
[{Red,Blue},Top],Filling -> {1->{2},2->{3}}]
Out[32]=
```



**Figure 5-26.** Linear Filled Gradient red, blue, line style

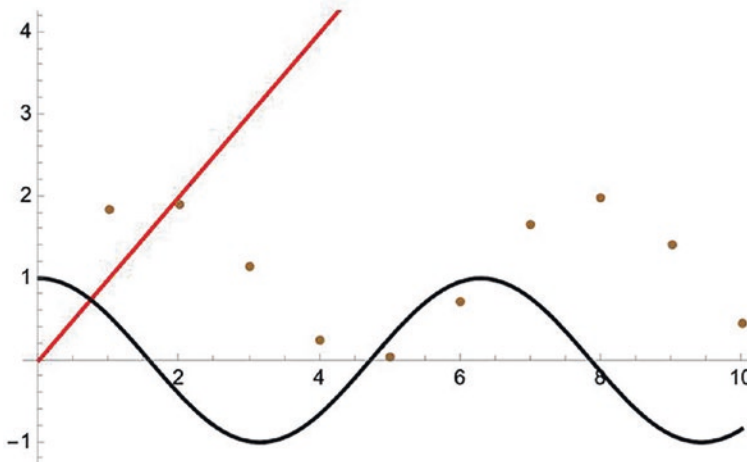
## Combining Plots

To display overlap graphics, there are ways to display the graphs even if they are not of the same type. The following example assigns names to plots without showing the result of each one and finally shows the three graphs. The `Show` command shows previously defined plots; the arguments are graphic objects followed by options. This is an alternative to doing multiple listable subplots.

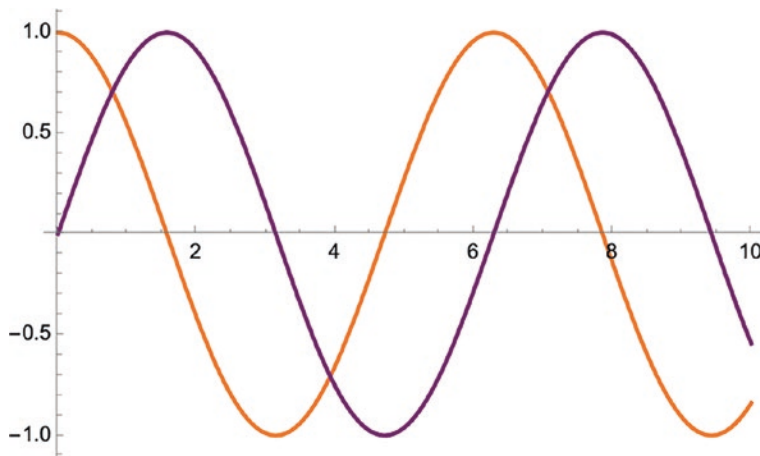
```
In[33]:= plot1=Plot[x,{x,0,10},PlotStyle->Red];
plot2=Plot[Cos[x],{x,0,10},PlotStyle->Black];
plot3=ListPlot[Table[Sin[i]+1,{i,1,10}],PlotStyle->Brown];
Show[plot1,plot2,plot3,PlotRange->Automatic]
Out[33]=
```

As shown in Figure 5-27, `Show` changes the appearance of the graphics; the order in which they are entered is preserved when displayed. Although making the graphics within `Show` is possible, you can add colors within the `Plot` command to distinguish the different graphs (see Figure 5-28).

```
In[34]:= Show[Plot[Cos[x],{x,0,10},PlotStyle->Orange],
Plot[Sin[x],{x,0,10},PlotStyle->Purple],PlotRange->Automatic]
Out[34]=
```



**Figure 5-27.** Combined plots shown in the same graphic

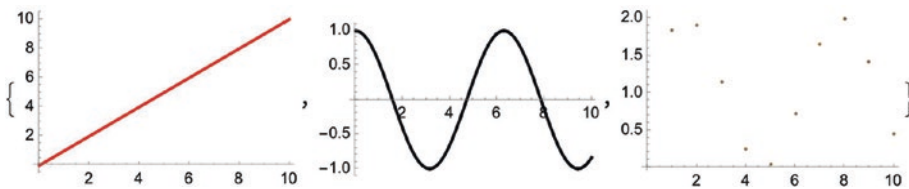


**Figure 5-28.** Cosine and Sine plot in the same graphic

There are several ways to create a list of graphs. You can assign variables to graphs and deploy them as a list.

```
In[35]:= {Plot1,Plot2,Plot3}
Out[35]=
```

As seen in Figure 5-29, these three graphs are separated by commas since it is a list.

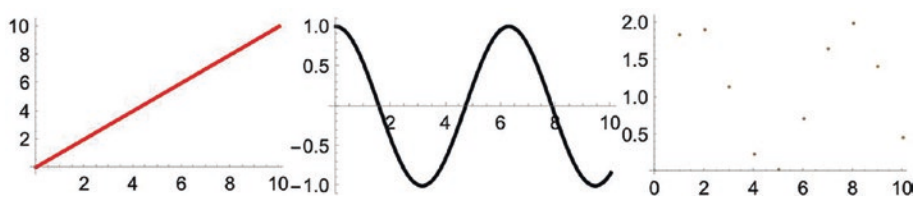


**Figure 5-29.** List of three different plots

## Multiple Plots

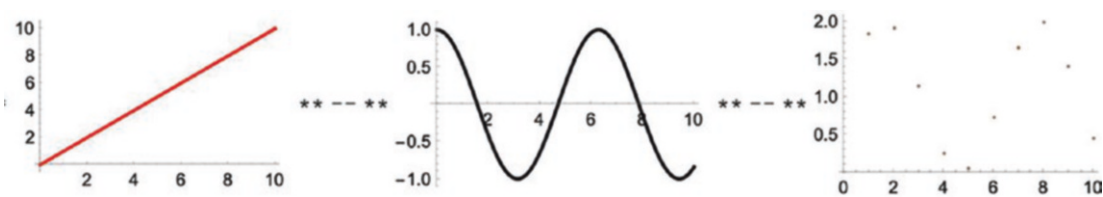
Multiple plots can be shown in a single output cell. To do this, use the Row command; this command allows the graphs to be displayed horizontally, with each graph on one side of the other (see Figure 5-30). However, Row generally displays expressions in row form, not just graphs.

```
In[36]:= Row[{plot1,plot2,plot3}]
Out[36]=
```



**Figure 5-30.** *Plots expressed as a row*

By entering a second argument for Row (see Figure 5-31), you have the option to add a separator between the graphs.

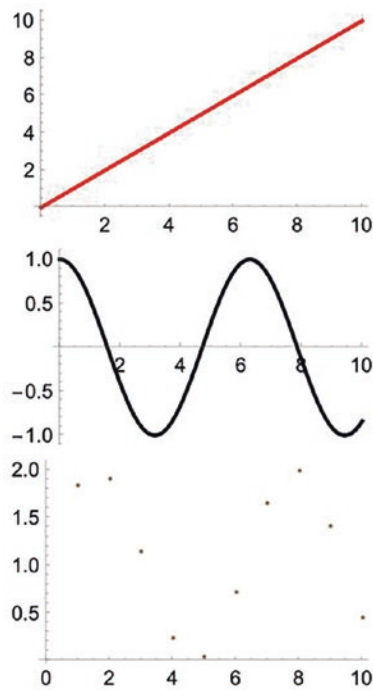


**Figure 5-31.** *Separator (\*\*--\*\*) added between each plot*

```
In[37]:= Row[{plot1,plot2,plot3},"**--**"]
Out[37]=
```

Alternatively, there is the Column command, which acts similarly to Row but displays expressions or graphs in column form (see Figure 5-32).

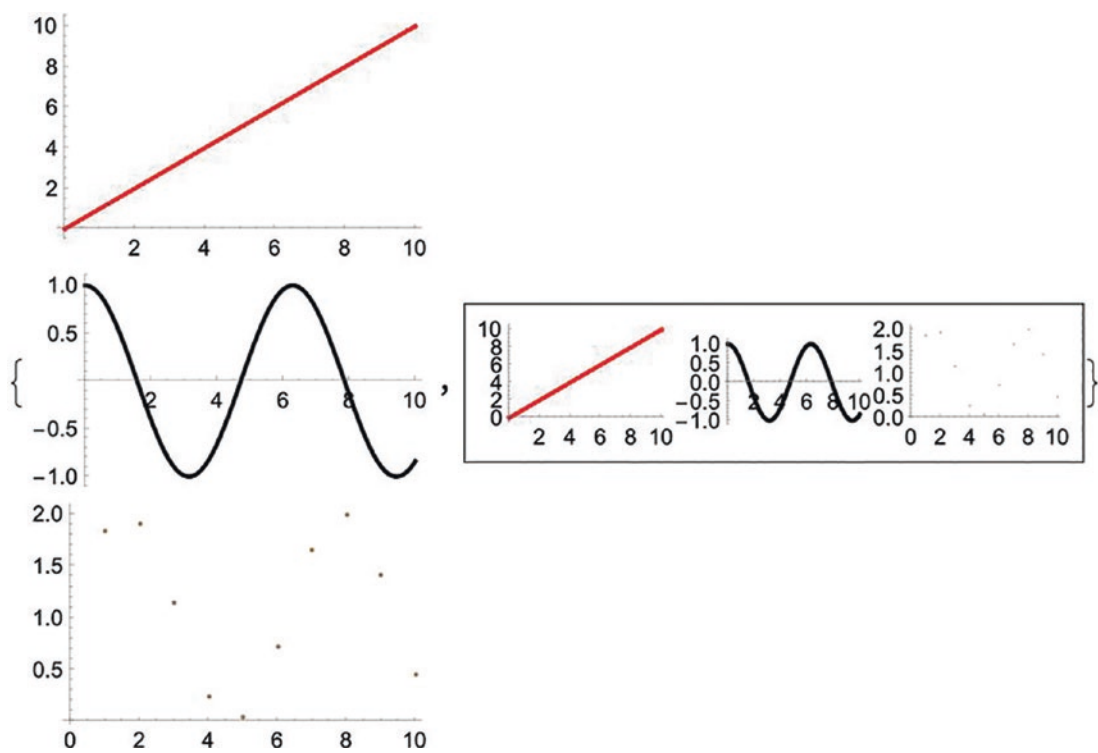
```
In[38]:= Column[{plot1,plot2,plot3}]
Out[38]=
```



**Figure 5-32.** Graphics expressed as a column

If you look at the following example, it is possible to add frames over the entire chart (see Figure 5-33) for both columns and rows.

```
In[39]:={Column[{plot1,plot2,plot3},Frame-> True],
Row[{plot1,plot2,plot3},Frame->True, FrameMargins->Medium]}
Out[39]=
```



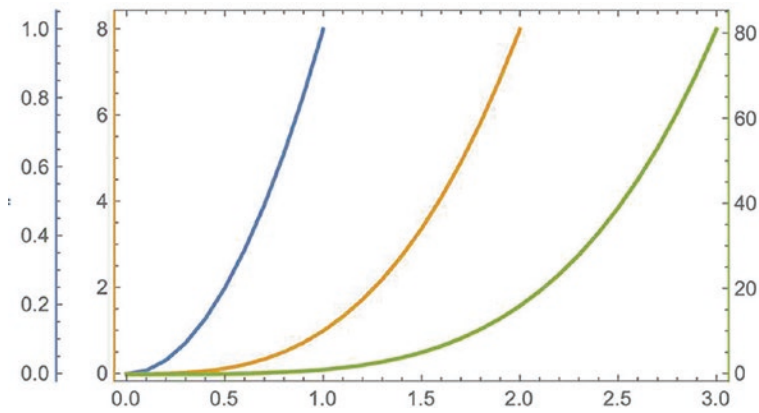
**Figure 5-33.** Exhibit of column and row expression for the three plots

## Multiaxis Plots

Since the new version, creating a single graph with multiple coordinate systems into a single pack requires linking the axes with different styles using `MultiAxisArrangement`. So, the curves connect through the same axis (see Figure 5-34).

```
In[40]:= ListLinePlot[{Table[{x,x^2},{x,0,1,0.1}],Table[{x,x^3}, {x,0,2,0.1}],
Table[{x,x^4},{x,0,3,0.1}]],MultiAxisArrangement-> All]
Out[40]=
```



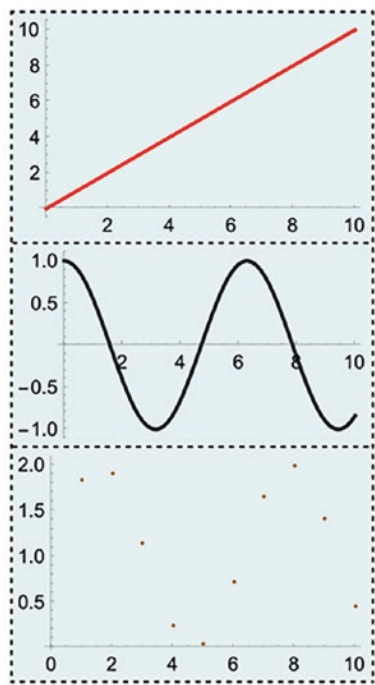


**Figure 5-34.** An exhibit of column and row expression for the three plots

## Coloring Plot Grids

Column and Row allow you to customize graphs. There are various ways of changing the color of the frame and adding shading to the graphs (see Figure 5-35).

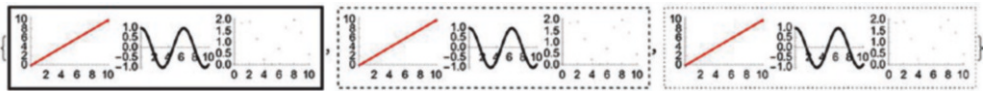
```
In[41]:= Column[{plot1,plot2,plot3},Frame->True,Background->LightCyan,
FrameStyle->Directive[Black,Dashed],Dividers->All]
Out[41]=
```



**Figure 5-35.** Column graphics with multiple features

Some options are available depending on whether you use a Row or Column. With Column, there is the option of dividers; in Row, there is no such option, but it is done via a separator, as you saw earlier. Using Table, it is possible to create different shapes on the graphs, either by color or frames, as shown in Figure 5-36.

```
In[42]:= Table[Row[{plot1,plot2,plot3},Frame->True,FrameStyle->Opts],
{Opts,{Thick,Dashed,Dotted}}]
Out[42]=
```

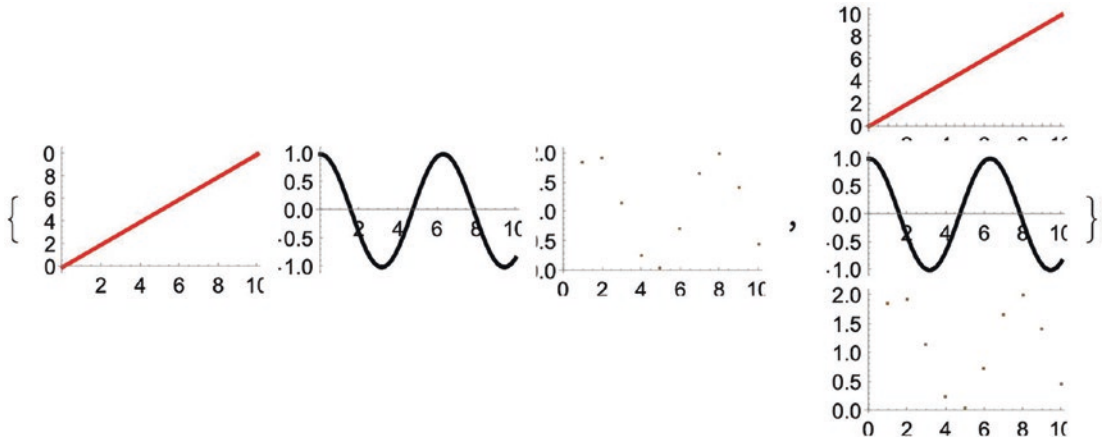


**Figure 5-36.** Table of multiple features implemented with the Row command

Next, let's address the existing alternative using `GraphicsRow` and `GraphicsColumn`. Around these commands, there are also options for the image size (see Figure 5-37).

```
In[43]:= {GraphicsRow[{plot1,plot2,plot3},ImageSize->Medium],
GraphicsColumn[{plot1,plot2,plot3},ImageSize->Small]}
```

Out[43]=

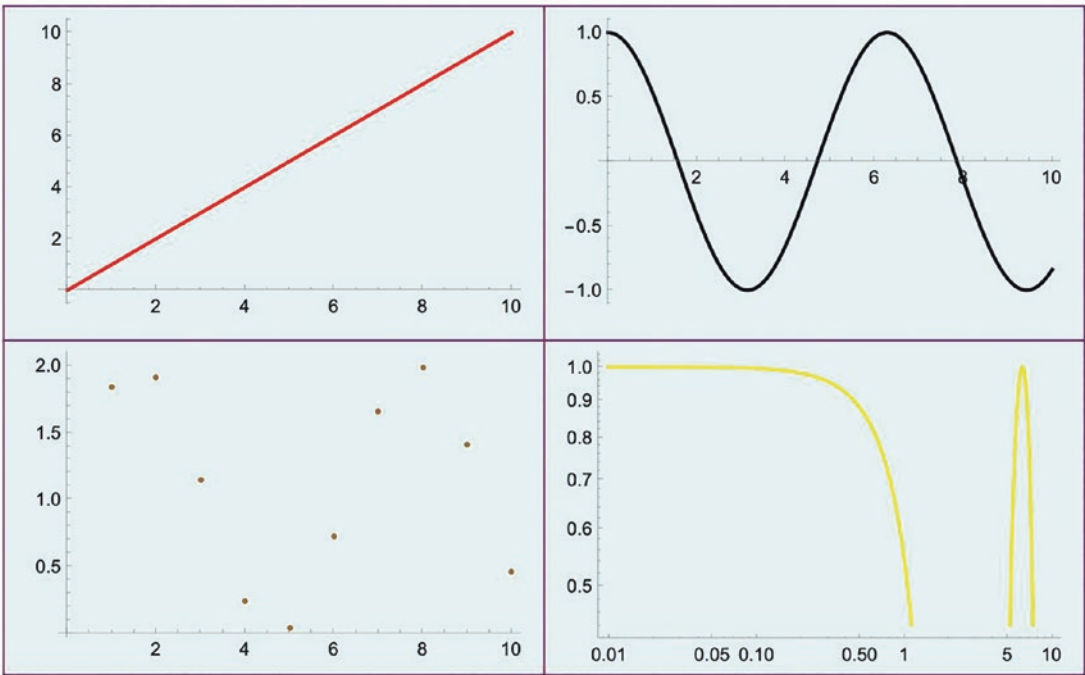


**Figure 5-37.** *GraphicsRow vs. GraphicsColumn*

`GraphicsRow` and `GraphicsColumn` are commands with specific shapes for constructing graphics, whether polygons, lines, dots, and so on. In addition, with `Rows` and `Columns`, the graphs are independent. With `GraphicsRow` or `GraphicsColumn`, if you select the graph, it is a unique image containing (in this case) the three plots you have made.

Another useful command shows you the graphs as a network, taking up the point stated earlier—if you select the graph, it is a unique image. The following example adds another chart to better illustrate why it's helpful to use `GraphicsGrid` (see Figure 5-38).

```
In[44]:= plot4=LogLogPlot[Cos[x],{x,0,10},PlotStyle->Yellow];
GraphicsGrid[{{plot1,plot2},{plot3,plot4}},Frame->All,FrameStyle->Purple,
Background->LightCyan]
Out[44]=
```



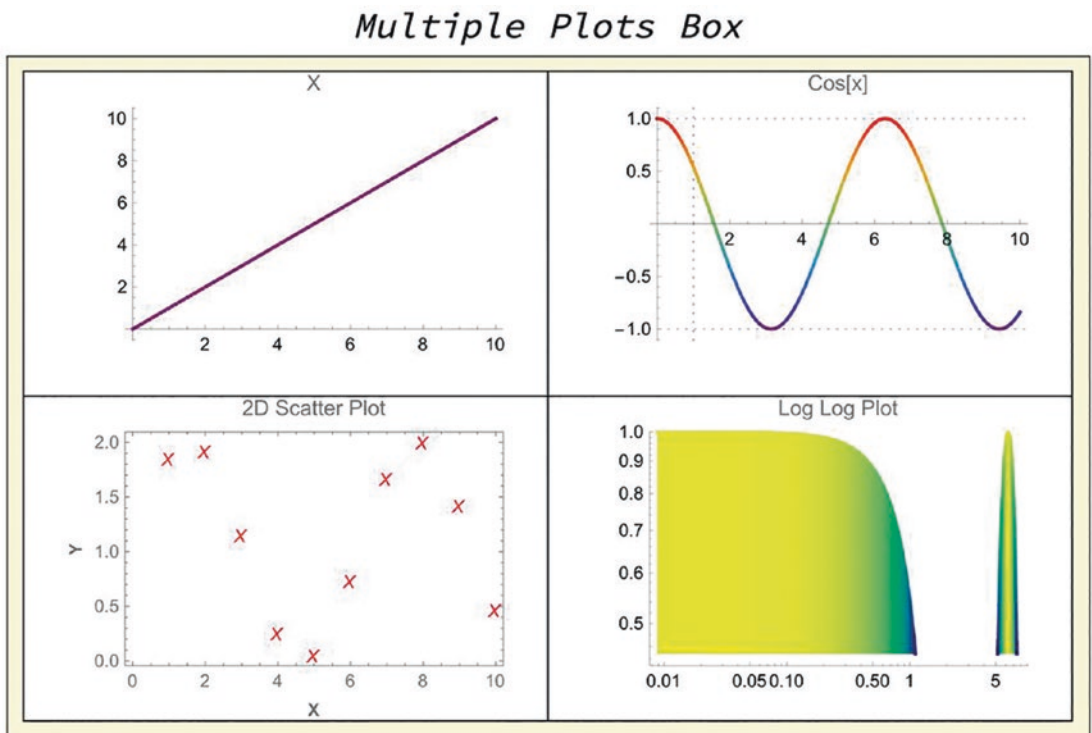
**Figure 5-38.** *GraphicsGrid showing four different plots*

As shown in Figure 5-38, this shape can help you compactly visualize four graphs at once. Without a doubt, the graphs do not have to be so simple. The options you have seen throughout this chapter can also be added, such as titles and labels on the axes, grid lines and colors, and more, as shown in the following example.

```
In[45]:=
newPlot1=Plot[x,{x,0,10},PlotStyle->{Purple,Thick},PlotLabel->"X"];
newPlot2=Plot[Cos[x],{x,0,10},GridLines->{{-1,0,1},{-1,0,1}},GridLinesStyle->Directive[Dotted,Blue],PlotLabel->"Cos[x]",ColorFunction->"Rainbow"];
newPlot3=ListPlot[Table[Sin[i]+1,{i,1,10}],Frame->True,FrameLabel->{Style["X",Bold],Style["Y",Bold]},PlotStyle->Red,PlotMarkers->"X",PlotLabel->"2D Scatter Plot"];
newPlot4=LogLogPlot[Cos[x],{x,0,9},Filling->Axis,ColorFunction->"BlueGreenYellow",PlotRange->{0,1},PlotLabel->"Log Log Plot"];
```

Now that you have the new plots, you can compare them by putting them as a nested list in GraphicsGrid (see Figure 5-39).

```
In[46]:=Labeled[GraphicsGrid[{{newPlot1,newPlot2},{newPlot3,newPlot4}},
Frame->All,Background->White,Spacings->1],Style["Multiple Plots
Box",20,Italic],Top,Frame->True,Background->LightYellow]
Out[46]=
```

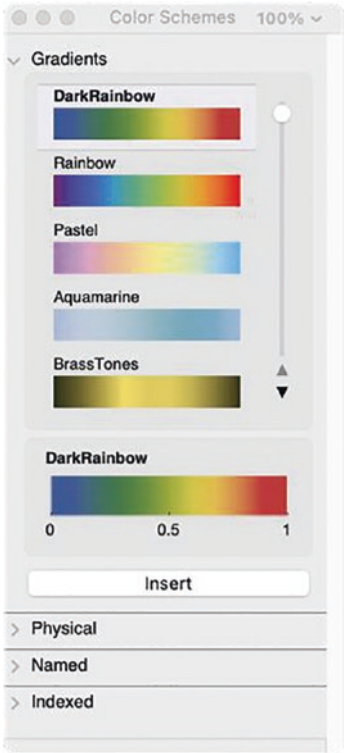


**Figure 5-39.** Grid of multiple plots

This is not restricted to displaying 2D graphs; it also applies to 3D graphs and other types of charts.

# Colors Palette

If you are interested in more colors, there is a gamma of various types of colors in Mathematica. For this, go to the menu in Palettes ► Color Schemes, as the color palette in Figure 5-40 shows.



**Figure 5-40.** Colors palette

The tabs that appear are of the colors associated with the different classes. To defer through the colors in the tabs, use the arrows, and the different names of the colors and their color or gradient are displayed. If you want to introduce colors that are not reserved words, then you use the insert button. For example, go to the Gradient tab and click the Insert button, which inserts the function with the chosen color into the notebook.

To illustrate, let’s look at the following example. Select the Color BrownCyanTones, insert it with the button, evaluate the expression, and get the result of the ColorDataFunction (see Figure 5-41).

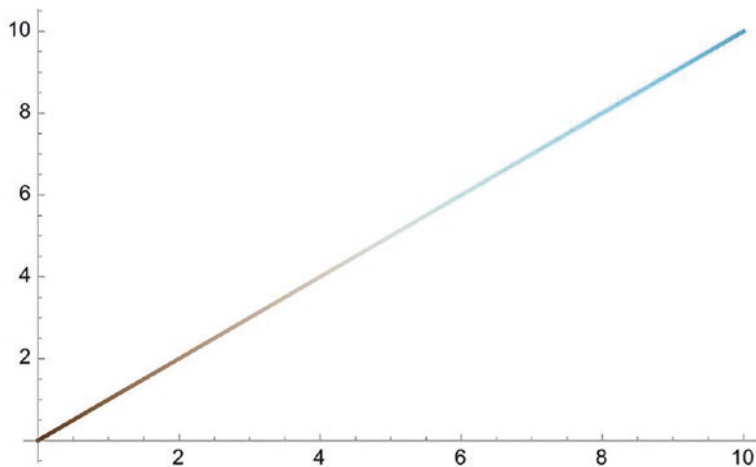
```
In[47]:= ColorData["BrownCyanTones"]
Out[47]=
```



**Figure 5-41.** *ColorData object*

This gives you a color data object showing the name, color type, class, and domain. Gradient colors are intricate in text and work best with the `ColorFunction` function. So now that you know the name, you can assign it a color (see Figure 5-42).

```
In[48]:= Plot[x,{x,0,10},ColorFunction->ColorData["BrownCyanTones"]]
Out[48]=
```



**Figure 5-42.** *Gradient color of straight line  $x$*

---

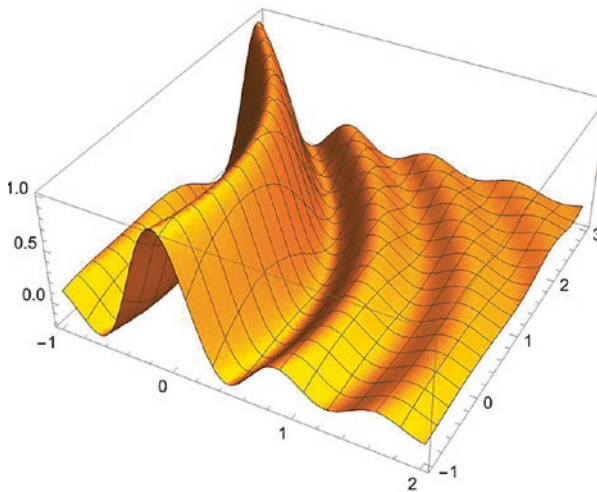
**Note** Plain colors are located in the named tab of the palette.

---

## 3D Plots

Mathematica can perform various types of 3D graphics, many of which are simple. 3D functions are displayed as surfaces in space. Figure 5-43 presents the example.

```
In[49]:= Plot3D[Sinc[x*8+y^2],{x,-1,2},{y,-1,3},ImageSize->Medium,  
PlotPoints->20]  
Out[49]=
```



**Figure 5-43.** 3D plot figure

Mathematica allows you to observe the graph by moving with the cursor. Hovering over the chart changes the cursor to rotating arrows, which means you can move the chart to observe it from different points. One last observation is that when you press the Ctrl or Cmd key, you can magnify the chart, keeping its position fixed.

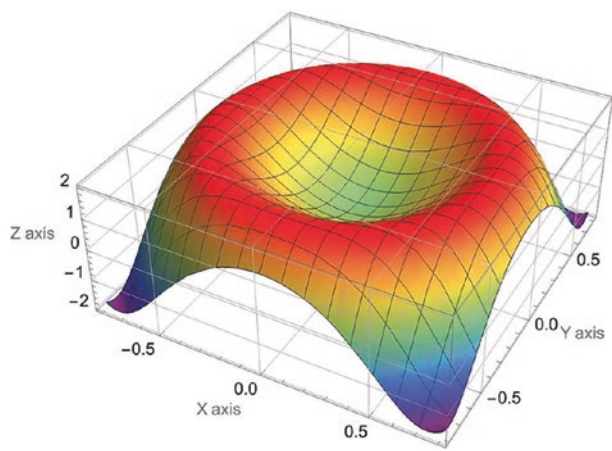
Note that the cursor can manipulate 3D graphs so that you can visualize the angle spread graph. Common standard Mathematica displays the graph as a mesh, which can be modified with the Mesh option, as you saw earlier, or by adding more points to evaluate with the PlotPoints option. This increases the number of points in both directions in both x and y. It also serves to improve the quality of the chart.



## Customizing 3D Plots

3D graphics can also be customized as 2D graphics (see Figure 5-44) as labels to axes, colors, grids, and so forth. Figure 5-44 shows a 3D plot with the AxesLabel, ColorFunction, and FaceGrids options.

```
In[50]:= Plot3D[Sin[4(x^2+y^2)]/0.5,{x,-0.8,0.8},{y,-0.8,0.8}, AxesLabel->
{"X axis","Y axis","Z axis"},ColorFunction->"Rainbow", FaceGrids->All]
Out[50]=
```



**Figure 5-44.** Gridded 3D plot

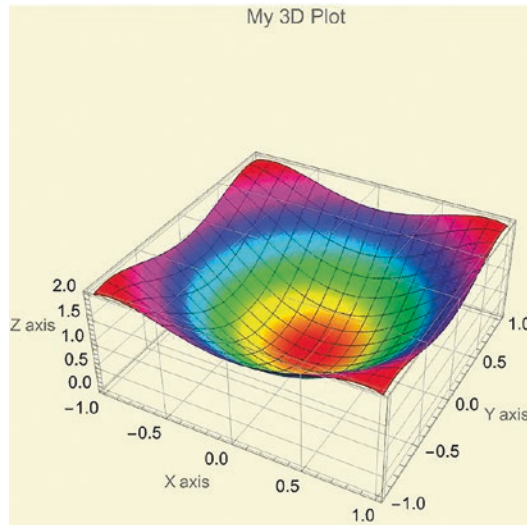
Table 5-1 shows general options for 3D graphics.

**Table 5-1.** Plot Options

Option	Instructions
AspectRatio	Height/width ratio
AxesLabel	Add text to axes
PlotStyle	Color, opacity, thickness, etc.
PlotRange	Range of values
PlotLabel	Plot title
Background	Background Color

Customization of graphics depends on how you plan to exhibit them. There is no limit on how graphics are presented. The following example plots a 3D function and colors the background light yellow (see Figure 5-45).

```
In[51]:= Plot3D[Sin[0.9(x^2+y^2)]/0.5,{x,-1,1},{y,-1,1},AxesLabel->
{"X axis","Y axis","Z axis"},FaceGrids->All,ColorFunction->Hue, PlotLabel->
"My 3D Plot",Background->LightYellow,ViewAngle->Pi/7]
Out[51]=
```

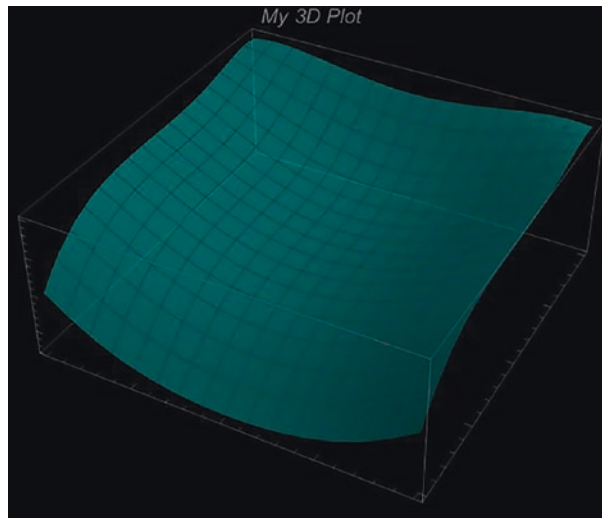


**Figure 5-45.** Customized 3D plot

## Hue Color Function and List3D

The Hue color function is a directive that specifies that the values are colored depending on the height they are at. There are three arguments for the Hue color function. The first is for the tone of the color (hue); the second marks the saturation; the third marks the bright one; and the fourth is the opacity. With hue, it is possible to adequately identify the high and low areas from a graph (see Figure 5-46) in the four previous features. You can mark these four different parameters. The hue parameters are in the range of 0 to 1.

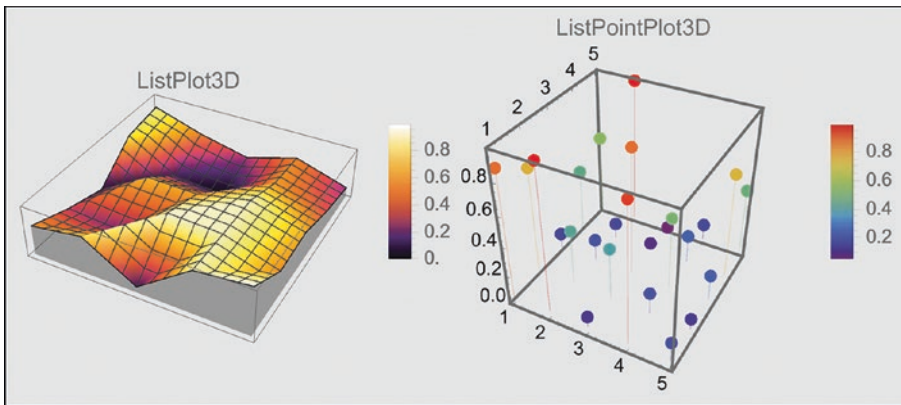
```
In[52]:= Plot3D[Sin[0.9(x^2+y^3)]/0.5,{x,-1,1},{y,-1,1}, FaceGrids->
None,ColorFunction -> (Hue[0.5,1,0.6,0.5]&),PlotLabel->Style["My 3D
Plot",Italic,"Arial"], Background->Black]
Out[52]=
```



**Figure 5-46.** 3D plot with colored Hue values

For 3D scatter plots (see Figure 5-47), you can do it using the same data. With `ListPlot3D`, the points are joined together to create a surface represented by the height values of each point. With `ListPointPlot3D`, a scatter plot is generated in 3D points.

```
In[53]:=Row[{ListPlot3D[Table[RandomReal[1,5],{i,5}],ColorFunction->
"SunsetColors",Ticks->None, PlotLegends-> BarLegend[Automatic,
LegendMarkerSize->90],ImageSize-> Small,PlotLabel->"ListPlot3D",Filling->
Bottom,BoxRatios-> Automatic] , ListPointPlot3D[Table[RandomReal[1,5],
{i,5}], ColorFunction->"Rainbow", PlotLegends->BarLegend[Automatic,
LegendMarkerSize->90], ImageSize->Small, PlotLabel->" ListPointPlot3D",
Filling->Bottom,BoxStyle->Thick, BoxRatios->{1,1,1}]],Background->Lighter
[Gray,0.80],Frame->True]
Out[53]=
```



**Figure 5-47.** *ListPlot3D* and *ListPointPlot3D* for random real numbers

## Contour Plots

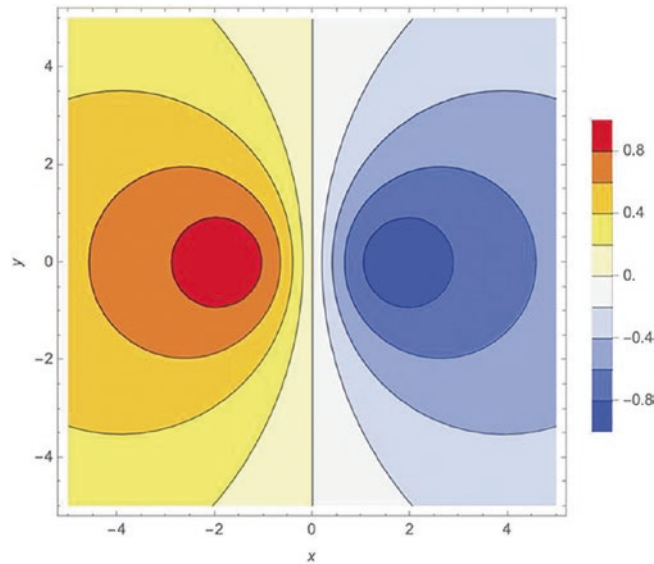
One way to visualize a two-variable function is to use a scalar field in which the scalar  $z = f(x, y)$  is mapped to the point  $(x, y)$ . A scalar field can be characterized by its contours (or contour lines) along which the value of  $f(x, y)$  is constant. The trace lines of contour line plots or contours can be done using the `ContourPlot` command, like in the next example.

```
In[54]:= ContourPlot[-((Pi*x)/(3+x^2+y^2)), {x, -5, 5}, {y, -5, 5}, ColorFunction->"Temperature", PlotLegends->Automatic, FrameLabel->{x, y}]
Out[54]=
```

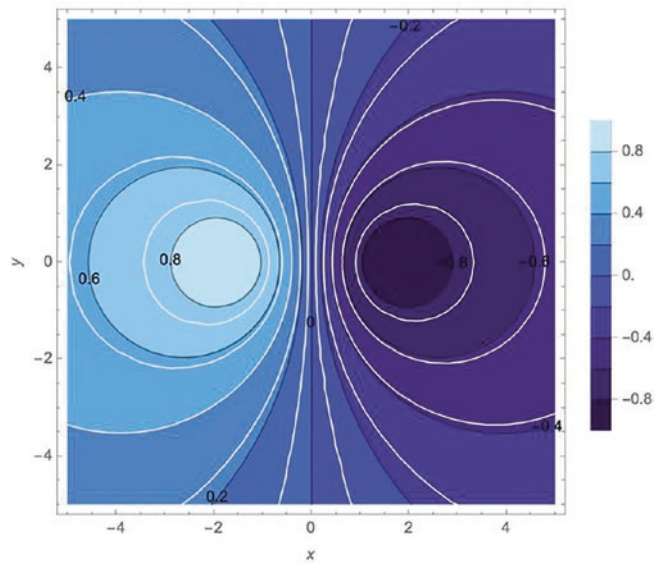
Figure 5-48 plots a contour plot using the `ColorFunction` and `PlotLegends` options. When you use `PlotLegends`, you specify what type of legends the chart should use; in this case, you use `automatic`. This shows you the scale of the contours depending on the color of each outline; for example, red is when it is at 0.8 or greater. When you pass the cursor through the contour curves, the value of that curve appears. To label the values of the contour curves in the graph image, add the `ContourLabels` option and assign the value to `true`, as shown in Figure 5-49. To add lines that pass through the graph, use the `GridLines` command, as you saw earlier, or use `Mesh`. `Mesh` can be joined with `MeshFunction` or `MeshStyle`.

```
In[55]:= ContourPlot[-((Pi*x)/(3+x^2+y^2)), {x, -5, 5}, {y, -5, 5},
ColorFunction->"DeepSeaColors", PlotLegends-> Automatic, FrameLabel->
```

```
{x,y}, ContourLabels->True, Mesh->{10,10}, MeshStyle->{White},
MeshFunctions-> {#3&}]
Out[55]=
```



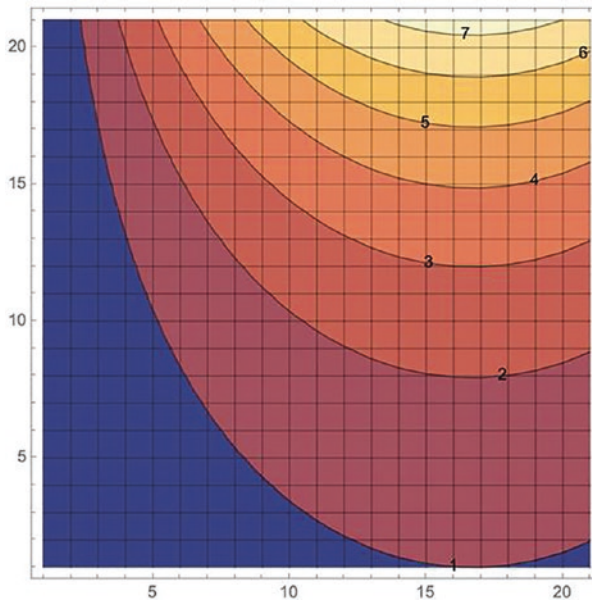
**Figure 5-48.** Contour plot for the defined  $z$  function



**Figure 5-49.** Contour lines added to the contour plot

To plot data into a contour plot (see Figure 5-50), use `ListContourPlot`. `ListContourPlot` creates a contour plot from an array of values shown in heights.

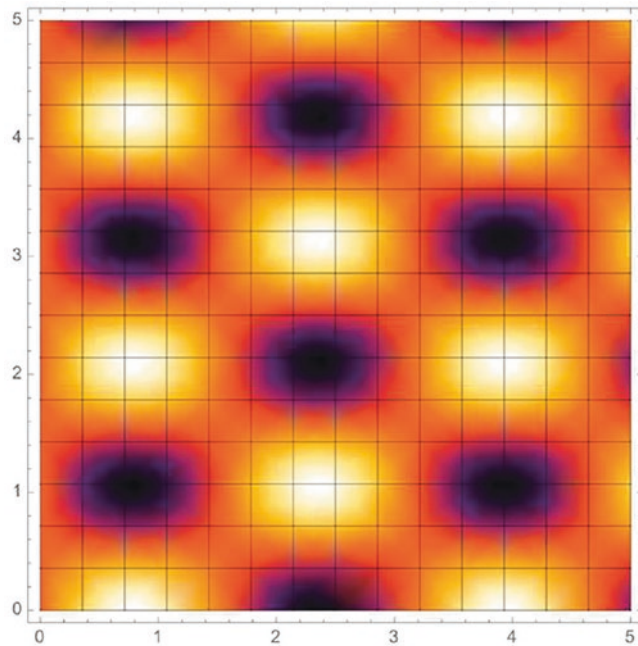
```
In[56]:= ListContourPlot[Table[Exp[x]*Sin[y],{x,0,2,.1},{y,0,2,.1}],
ContourLines->True,Mesh->Full,ContourLabels->True]
Out[56]=
```



**Figure 5-50.** *ListContourPlot*

Another plot is `DensityPlot` (see Figure 5-51). `DensityPlot` works similarly to `ContourPlot`.

```
In[57]:= DensityPlot[(Sin[2x]*Cos[3y])/5,{x,0,5},{y,0,5}, ColorFunction->
"SunsetColors", Mesh->Full]
Out[57]=
```

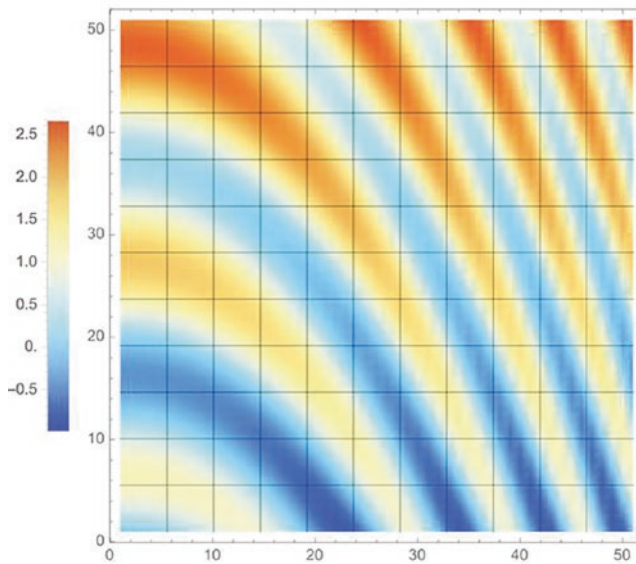


**Figure 5-51.** Density plot

You can plot density plots from data with `ListDensityPlot` (see Figure 5-52).

```
In[58]:= ListDensityPlot[Table[x/3 + Sin[3 x + y^2], {x, 0, 5, 0.1}, {y, 0, 5, 0.1}], ColorFunction -> "LightTemperatureMap", Mesh -> 10, PlotLegends -> Placed[Automatic, Left]]
Out[58]=
```





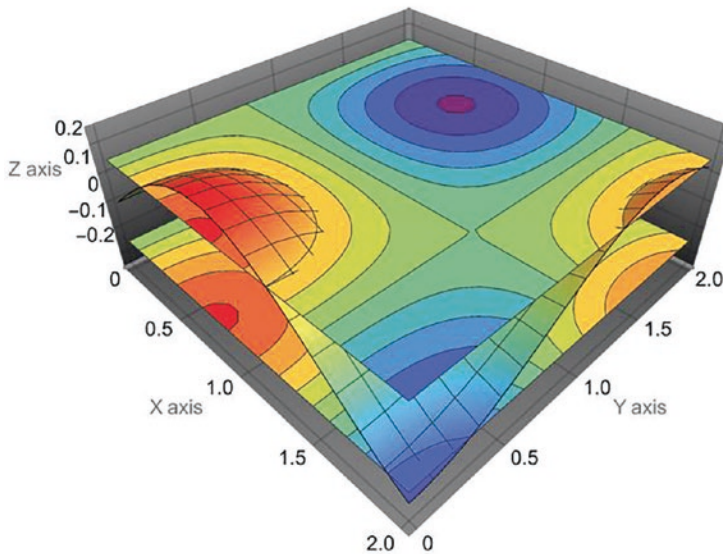
**Figure 5-52.** Data represented as a density plot

## 3D Plots and 2D Projections

With the Wolfram Language, it is possible to plot functions in 3D and, at the same time, project the contour maps to planes as the axis, as shown in Figure 5-53.

```
In[59]:= Show[Plot3D[(Sin[2 x]*Cos[2 y])/4, {x, 0, 2}, {y, 0, 2},
PlotStyle -> Directive[Opacity[1]], AxesLabel -> {"X axis", "Y axis", "Z
axis"},ColorFunction -> "Rainbow", PlotTheme -> "Marketing"],SliceContourPlo
t3D[(Sin[2 x]*Cos[2 y])/4, {z == -0.15, z == 0.15}, {x, 0, 2}, {y, 0, 2},
{z, -1, 1}, ColorFunction -> "Rainbow", Boxed -> False], ViewPoint ->
{1, -1, 1}]
Out[59]=
```





**Figure 5-53.** 3D plot with contour plots along the *xy* plane

Let's discuss what happens in the code. You plot a function in 3D (see Figure 5-53), and to this function, you add color, using the command directive to define the type of opacity, which is set to 1. This is followed by typing the name of the corresponding axes for the *x*, *y*, and *z* axes. The `ColorFunction` option can help define a function for the color type; in this case, it is `Rainbow`. The `PlotTheme` is an option to plot with various themes for visualization. Coming to this point, you move on to the `SliceContourPlot3D`, which gives you a graph of the function, either on a plane or a surface. you have plotted when *z* is worth  $\pm 0.15$ . A cut is made on the *xy* plane. This occurs when *x* and *y* are in the range of 0 to 2, and *z* is in the range of -1 to 1. In the end, you combine the two graphs with the `Show` command; you use this command because you would not have the function's graph in 3D only by plotting on its slice contour plot.

## Plot Themes

Preconstructed themes can be accessed using the `PlotTheme` option. You see the autocomplete menu when you add the `PlotTheme` option, followed by the first apostrophe. Figure 5-54 shows the different themes that exist.



Figure 5-54. PlotTheme pop-up menu

PlotTheme supports 3D plots, as shown in Figure 5-55.

```
In[60]:= data=Flatten[Table[{x,y,Sin[10(x^2+y^2)]/10},
{x,-2,2,0.2},{y,-2,2,0.2}],1]; ListPointPlot3D[data,ColorFunction->
"LightTemperatureMap", PlotTheme->"Detailed",ViewPoint->{0,-2,0},
ImageSize->250,PlotLegends->Placed[BarLegend[Automatic, LegendMarkerSize->
90],Left], ImageSize->20]
Out[60]=
```

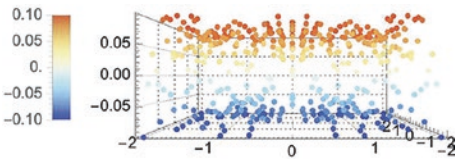
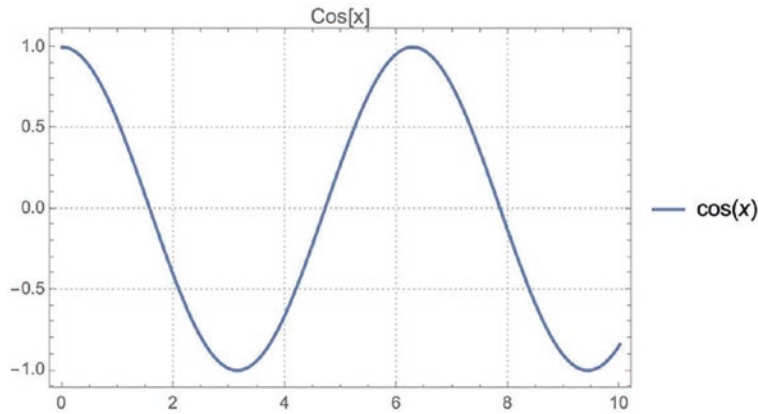


Figure 5-55. 3D scatter plot

These themes can be used for both 2D and 3D graphics. Now, let's look at another type of theme for a two-dimensional chart (see Figure 5-56).

```
In[61]:= Plot[Cos[x],{x,0,10},PlotLabel->"Cos[x]",PlotTheme->"Detailed"]
Out[61]= cos(x)
```

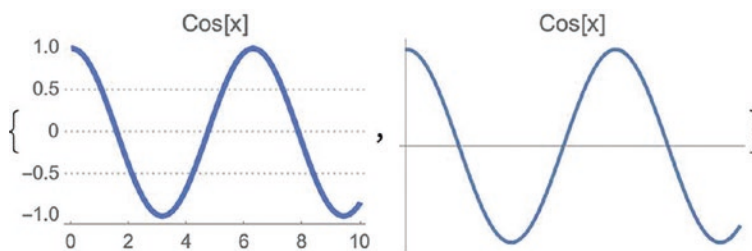


**Figure 5-56.** 2D plot theme: Detailed

Let's discuss a characteristic of PlotTheme. Some themes already have functions within these themes. Figure 5-55 shows that the Detailed theme adds frames, plot legends, and grid lines, even though you can add them manually.

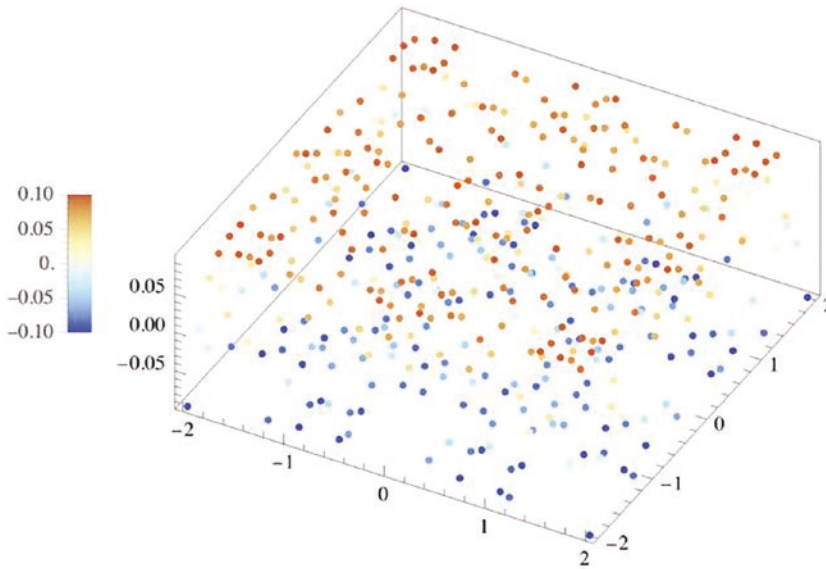
It is also notable that other topics can only be used for explanatory and demonstrative purposes—that is, no extra information is needed on the chart, but you need to be able to express the information effectively and concretely, as in the Business and Minimal themes (see Figure 5-57).

```
In[62]:= Table[Plot[Cos[x],{x,0,10},PlotLabel->"Cos[x]",PlotTheme->P1],{P1,
{"Business","Minimal"}}]
Out[62]=
```



**Figure 5-57.** Business and Minimal plot themes

While there are also topics that show more details, like the Detailed theme you saw earlier, other themes exist, like the Scientific theme, as shown in Figure 5-58. You can add more options, such as `ColorFunction` and a view, with the `ViewProjection` option, which allows you a fixed observation point.



**Figure 5-58.** Orthographic point of view

---

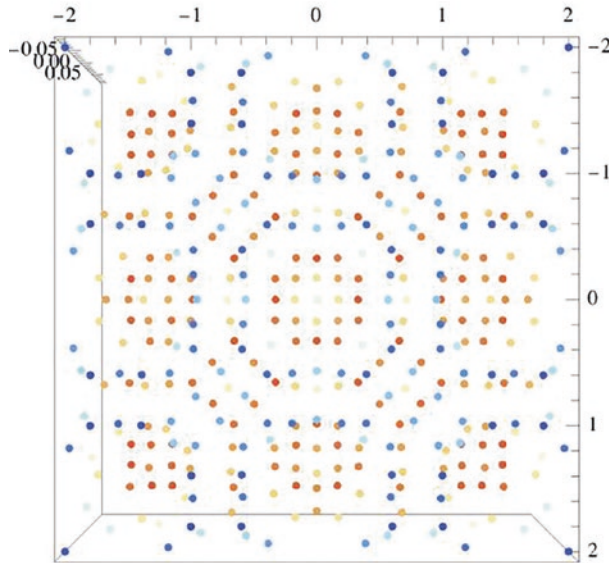
**Note** `PlotLegends` can work together with `ColorFunction`, displaying how the colors of the dots transition between blue and red, from lowest to highest.

---

```
In[63]:= data=Flatten[Table[{x,y,Sin[10(x^2+y^2)]/10},
{x,-2,2,0.2},{y,-2,2,0.2}],1]; ListPointPlot3D[data,ColorFunction->
"LightTemperatureMap",PlotLegends-> Placed[BarLegend[Automatic,
LegendMarkerSize->90],Left], PlotTheme->"Scientific", ViewProjection->
"Orthographic"]
Out[63]=
```

If you want to observe through the coordinate measurements, use the `Viewpoint` option, which is governed by {x coordinate, y coordinate, z coordinate}. These coordinates are relative to the graph's center, as Figure 5-59 shows.

```
In[64]:= ListPointPlot3D[Data,ColorFunction->"LightTemperatureMap",
PlotLegends->Automatic,PlotTheme->"Scientific",ViewPoint->
{0,0,-2},ImageSize->Medium]
Out[64]=
```



**Figure 5-59.** Viewpoint for  $x$  and  $y$  equal 0 and  $z$  equal -2

## Summary

This chapter introduced the basics of data visualization, emphasizing 2D plots, plotting data, and user-defined functions. As progress is made, the section on customizing plots covers text to charts, frames, grids, and filled plots, including further content on fill patterns and gradient filling, followed up by discussing how plot combinations are done, focusing on multiple plots, and coloring plot grids and concentrating on new additions like multi-axes plots. Furthermore, an overview of the color palette was presented, followed by a segmentation of 3D Plots, elaborating on the customization, Hue coloring, and contour plots. Finally, it culminates with an outlook on the variety of plot themes for 3D graphs.

## CHAPTER 6

# Statistical Data Analysis

This chapter reviews concepts and techniques to analyze with the Wolfram Language, perform a linear adjustment through equations, and implement specialized functions of the Wolfram Language for the same purpose, using statistical functions. The Wolfram Language is a useful tool for statistics and probability. Mathematica has the functions to perform numerical and approximate calculations for descriptive statistics and random distributions, random numbers, and random sampling methods, as you see in this section.

## Random Numbers

This section reviews the basic commands to generate random numbers—for the case of integers, real and complex. You see the functions of performing random sampling with replacement and without replacement and, in addition, ensuring that the results are reproducible for random numbers.

To create random numbers, there are several functions to generate random integers and real ones. The `RandomInteger` function generates entered random numbers; if no arguments are entered in the function, the generation interval is 0 or 1.

```
In[1]:= RandomInteger[]
```

```
Out[1]= 0
```

To enter a range, you must define it within the function; for example, between -1 and 1.

```
In[2]:= RandomInteger[{-1,1}]
```

```
Out[2]= 1
```

To generate a list of random numbers, you must define how many numbers you want within the list.

```
In[3]:= RandomInteger[{-1,1},7]
Out[3]= {-1,0,1,1,1,1,1}
```

To repeat the numbers, add the form of the list or nested list as a second argument. For example, create a nested list of seven total items in each sublist with four items.

```
In[4]:= RandomInteger[{-10,10},{7,4}]
Out[4]= {{-8,7,7,0},{-4,-8,10,-8},{10,8,-8,0},{-2,-6,8,-10},
{8,-1,-6,-4},{1,4,0,-1},{5,7,9,10}}
```

The function for generating random numbers with a decimal point is called `RandomReal`. It works similarly to `RandomInteger`, where the interval is between curly braces.

```
In[5]:= RandomReal[]
Out[5]= 0.020413
```

A command for complex random and prime numbers also exists.

```
In[6]:= RandomComplex[]
Out[6]= 0.727318 +0.998602 I
```

You must define a minimum and maximum interval for random prime numbers—for example, if it is a prime number of the first 100.

```
In[7]:= RandomPrime[{1,100},6]
Out[7]= {89,2,59,71,53,29}
```

This type of function generates pseudorandom numbers so that you can set a seed to generate the numbers. This is done with `SeedRandom`. With a seed, you can ensure that the starting sequence of random numbers generated is the same to make random outputs reproducible. To set a seed, use the `SeedRandom` command. The following example sets a seed followed by a sequence of random numbers; once the seed is introduced, the results should be the same for that seed.

```
In[8]:= SeedRandom[6467789];RandomInteger[{-1,1},3]
Out[8]= {0,1,0}
```

The seed must go in the same code block to generate the results. There is the option to choose the method. The following example uses the MersenneTwister method, which generates random numbers. Using another method allows you to generate sequences of different random numbers.

```
In[9]:=SeedRandom[Method->"MersenneTwister"];RandomInteger[{-1,1},{3,3}]
// MatrixForm
Out[9]//MatrixForm
```

$$\begin{pmatrix} -1 & -1 & -1 \\ 0 & 0 & -1 \\ 0 & 0 & 1 \end{pmatrix}$$

The seed enters the function without arguments to return to the original value.

```
In[10]:= SeedRandom[];
```

In addition to introducing a seed, you can create blocks of random numbers in which functions can be used locally and not affect random behavior outside these blocks. This is done with the BlockRandom function.

```
In[11]:= BlockRandom[RandomReal[1]]
Out[11]= 0.774569
```

If you run an algorithm that produces random numbers within the BlockRandom and declare the seed, this should not impact other processes where random numbers are generated outside the BlockRandom. To illustrate, let's look at the example.

```
In[12]:= SeedRandom[121];
{RandomReal[],BlockRandom[RandomReal[]],RandomReal[],RandomReal[]}
Out[13]= {0.994955,0.788549,0.788549,0.957081}
```

As seen, the latter process generated different random numbers

## Random Sampling

Use the RandomChoice function to make a sample with a replacement. To select a single item, you write only the list. You set a seed to get the same results.



```
In[14]:= SeedRandom[12345]; ranData=RandomReal[{0,1},10]
Out[15]={0.158069,0.599452,0.656143,0.918006,0.0805897,0.682397,0.638187,
0.431772,0.126333,0.973705}
```

This generated a list of 10 random numbers from 0 to 1, and now you randomly choose an item of these numbers.

```
In[16]:= RandomChoice[ranData]
Out[16]= 0.973705
```

This gives you a single result from the list of 10 items. Similarly, you can choose the number of samples with some elements, with the following form: `RandomChoice[“data,” “number of samples,” “several elements”]`. You now pick three samples with one element of the ten elements.

```
In[17]:= RandomChoice[ranData,{3,1}]
Out[17]= {{0.126333},{0.431772},{0.973705}}
```

Although, if you want it in the same sample, you only need to specify the number of elements to choose from.

```
In[18]:= RandomChoice[ranData,5]
Out[18]= {0.0805897,0.158069,0.158069,0.0805897,0.973705}
```

To get a sampling without replacement, use `RandomSample`. This function only chooses a list item from the data list once. To choose, you only specify the number of elements in the sample as the second argument since the first one corresponds to the data list.

```
In[19]:= RandomSample[ranData,9]
Out[19]={0.158069,0.682397,0.431772,0.599452,0.918006,0.638187,0.656143,
0.126333,0.0805897}
```

Looking at the details, you notice that there is no repeated value. Each item in the list is equally likely to be selected in sampling.

In the case that each item in the list has a specific weight associated with it, then to enter those terms, you use the following form of expression, `{w1, w2, w3...} → {element1, element2, element3...}`; the list of items is associated with a specific weight for replacement sampling. You denote the list of weights and do the sampling by associating the weights and elements.

```
In[20]:=w={0.03`,0.08`,0.22`,0.04`,0.12`,0.3`,0.12`,0.03`,0.04`,0.02`};
RandomChoice[w->ranData,2]
Out[20]= {0.656143,0.638187}
```

They are chosen depending on how each element is assigned a weight. For sampling without replacement, the process is analogous.

```
In[21]:= RandomSample[w->ranData,3]
Out[21]= {0.682397,0.656143,0.599452}
```

## Systematic Sampling

To perform a system sampling, you must determine the sample size,  $M$ . To get the sample size, you can list the items in the list or get the length of the list. To get started, you create a list of 200 prime numbers.

```
In[22]:= SeedRandom[09876]; rPrime=RandomPrime[{1,100},200];
Length[rPrime]
Out[24]= 200
```

The sample size was already calculated, so you must determine the size of a specific sample; for this case, you want a sample of 20 elements. Once the sample is determined, you calculate the interval of the denoted sampling  $j$ ;  $j$  is calculated through a ratio, the original sample size divided by the total number of elements in the specified sample.

```
In[25]:= j=Length[rPrime]/20
Out[25]= 10
```

This means that the sampling interval for the new sample is from 1 to 10. From here, you select a random number within the interval, and from there, you add  $j$  times to choose the next element; that is, for the first element, it is a random  $h$  number of the range  $[1,10]$ , for the second it is  $h + j$ , and for the third  $h + 3j$ , and so on, until it reaches the size of the original sample.

You chose a random number between 1 and 10.

```
In[26]:= RandomSample[Range[10],1]
Out[26]= {6}
```

The result means that you select from the sixth element. You deploy the list to have a better view of the data.

```
In[27]:= rPrime
Out[27]={7,41,3,7,83,61,41,29,89,5,17,3,41,73,73,67,29,71,23,13,31,19,89,
41,79,19,47,83,13,73,37,67,59,29,13,17,83,43,17,71,89,11,71,23,29,37,89,3,
89,11,41,59,2,37,41,31,59,79,61,13,59,53,53,59,2,43,11,73,41,37,3,31,13,
83,83,3,31,5,37,2,89,23,2,37,23,3,79,17,47,71,79,13,47,13,17,41,71,73,2,
53,29,7,2,7,79,97,83,31,3,43,29,11,37,67,11,41,67,13,23,2,59,53,89,61,29,
19,29,13,11,7,61,71,59,53,5,71,13,43,67,2,73,2,5,67,83,53,11,7,61,71,7,11,
83,59,47,67,17,83,43,53,17,59,11,11,61,2,11,97,2,73,41,7,41,19,41,71,53,3,
3,41,29,5,73,53,79,43,13,19,29,2,73,67,29,41,13,3,43,23,59,89}
```

To get the positions of the items to be selected, it would be the random number for the selection, which is 6, plus  $n$  times  $j$  until you have 20 elements.

```
In[28]:= Table[6+n*j,{n,0,19}]
Out[28]= {6,16,26,36,46,56,66,76,86,96,106,116,126,136,146,156,166,
176,186,196}
```

---

**Note** Remember that the position index starts from 1 to  $n$  elements.

---

You must choose the positions shown in the previous output. To choose, you use the double square bracket notation.

```
In[29]:= Table[rPrime[[6+n*j]],{n,0,19}]
Out[29]= {61,67,19,17,37,31,43,3,3,41,97,41,19,71,53,67,2,71,43,3}
```

Let's take a closer look at the selected elements, highlighting them in red (here it is plaintext) with the help of MapAt and Style.

```
In[30]:= MapAt[Style[#,FontColor-> ColorData["HTML"]["Red"]]&,
RPrime,{#}&/@{61,67,19,17,37,31,43,3,3,41,97,41,19,71,53,67,2,71,43,3}]
Out[30]={7,41,3,7,83,61,41,29,89,5,17,3,41,73,73,67,29,71,23,13,31,19,89,
41,79,19,47,83,13,73,37,67,59,29,13,17,83,43,17,71,89,11,71,23,29,37,89,3,
89,11,41,59,2,37,41,31,59,79,61,13,59,53,53,59,2,43,11,73,41,37,3,31,13,
83,83,3,31,5,37,2,89,23,2,37,23,3,79,17,47,71,79,13,47,13,17,41,71,73,2,
```

53, 29, 7, 2, 7, 79, 97, 83, 31, 3, 43, 29, 11, 37, 67, 11, 41, 67, 13, 23, 2, 59, 53, 89, 61, 29, 19, 29, 13, 11, 7, 61, 71, 59, 53, 5, 71, 13, 43, 67, 2, 73, 2, 5, 67, 83, 53, 11, 7, 61, 71, 7, 11, 83, 59, 47, 67, 17, 83, 43, 53, 17, 59, 11, 11, 61, 2, 11, 97, 2, 73, 41, 7, 41, 19, 41, 71, 53, 3, 3, 41, 29, 5, 73, 53, 79, 43, 13, 19, 29, 2, 73, 67, 29, 41, 13, 3, 43, 23, 59, 89}

As you can see, system sampling does not create a completely random sample. The random selection process comes in the first part when you select the first item to create the new sample. Once the first item is selected, the other selections are from a succession of non-random numbers. Another aspect to consider is the order of the original sample; if the elements are periodic, this can lead to significant variability in the selection of components.

## Commons Statistical Measures

Grasping the commonly used statistical formulas is crucial to understanding how the data behaves on a given set of conditions. Descriptive statistics are implemented once data has been collected, and it is one of the first steps in the process of exploratory data analysis, which allows you to find insights into the data collected in terms of discovering patterns, anomalies, trends, seasonality, variations, and so forth.

Exploratory data analysis is a set of techniques to detect characteristics that are not visible at first sight or revealed once the data has been collected. The basic structure of this technique relies on numeric data analysis, graphical representation, and a statistical model. Many reasons to use data exploratory analysis include reviewing for missing data, describing a general and particular idea of the underlying structure, and analyzing for different assumptions associated with the model creation, among many more.

The proposal for such a process was introduced by Jhon Tukey in 1977. To review this technique in more depth, visit the following reference, *Exploratory Data Analysis* (Tukey, J. W. [1977], Vol. 2, pp. 131-160).

## Measures of Central Tendency

Given a sample of data, you can calculate the descriptive measures. Central trend measures are those parameters that give you information on the average data values to be studied. The mean, also known as arithmetic mean, is a parameter calculated from

the sum of the values of the sample and divided by the sum of the number of elements. The Mean function calculates the average.

```
In[31]:= list1=Table[Prime[i],{i,10}];
"Prime list : "<>ToString@list1
"Mean: " <>ToString@Mean@N@list1
Out[32]= Prime list : {2, 3, 5, 7, 11, 13, 17, 19, 23, 29}
Out[33]= Mean: 12.9
```

---

**Note** The symbol <> is the short notation for StringJoin.

---

The median is the value that divides the sample into two equal parts; since it is the data's midpoint, the median is the symmetry value relative to the amount of data. The Median function gives you this value.

```
In[34]:= "Median: " <>ToString@Median@list1
Out[34]= Median: 12
```

Mode is the most common value of the sample. You use the Counts command, which gives you the number of occurrences of each item in the list.

```
In[35]:= Counts[list1]
Out[35]= <|2->1,3->1,5->1,7->1,11->1,13->1,17->1,19->1,23->1,29->1|>
```

In this case, the occurrence is 1. There are no repeated values; you can say there is no mode in this data sample.

## Measures of Dispersion

Dispersion measurements reveal information on the variability presented in the sample. The range tells you about the interval in which the data varies. This is taken by subtracting the max value and the minimum value. The Max and Min functions return a list's maximum and minimum values.

```
In[36]:= "Range: " <>ToString[Max[list1]-Min[list1]]
Out[36]= Range: 27
```

Variance is a measure obtained by subtracting the mean of each element in the sample. The result is squared, followed by adding the elements together. The summation is divided by the size of the sample. Its function is `Variance`.

```
In[37]:= "Variance: " <> ToString[N[Variance[list1],3]]
Out[37]= Variance: 81.4
```

Standard deviation is a measurement obtained from the square root of the variance or employing the `StandardDeviation` function.

```
In[38]:= {"Square root of Variance: " <> ToString[N[Sqrt[Variance[list1]],
2]], "StandardDeviation: " <> ToString[N[StandardDeviation[list1], 2]]}
Out[38]= {Square root of Variance: 9.0, StandardDeviation: 9.0}
```

The standard score,  $z$ , is a score that measures how many standard deviations are away from the arithmetic average for each sample element. The mathematical equation is  $z = \frac{x - \mu}{\sigma}$ , where  $x$  is the measure,  $\mu$  the mean, and  $\sigma$  the standard deviation. If  $z$  is positive, the element is greater than the mean. When  $z$  is negative, it is the opposite case. You determine the  $z$ -score for the second item in the list.

```
In[39]:= z=N[(list1[[2]]-Mean@list1)/StandardDeviation@list1,3];
"z score: " <> ToString@z
Out[40]= z score: -1.10
```

This result means that the score for the second element is 1.10 times below average.

Quartile calculation divides data into four equal parts. The lower quartile corresponds to the 25% quartile of the data, while the second quartile is 50%, the third quartile (the upper quartile) is 75%, and the fourth quartile (100%). To calculate the quartiles, you use the `Quartiles` function, which gives the values of the first, second, and third quartiles.

```
In[41]:= "Quartiles: " <> ToString@Quartiles[list1]
Out[41]= Quartiles: {5, 12, 19}
```

If you want to get a single value, use the `Quantile` function, followed by the percentile, to be calculated. Then, use the following for calculating the third quartile (75th percentile).

```
In[42]:= Quantile[list1,0.75]
Out[42]= 19
```

To calculate the interquartile range, which is the difference between the upper and lower quartiles, use the `InterquartileRange` function.

```
In[43]:= InterquartileRange[List1]  
Out[43]= 14
```

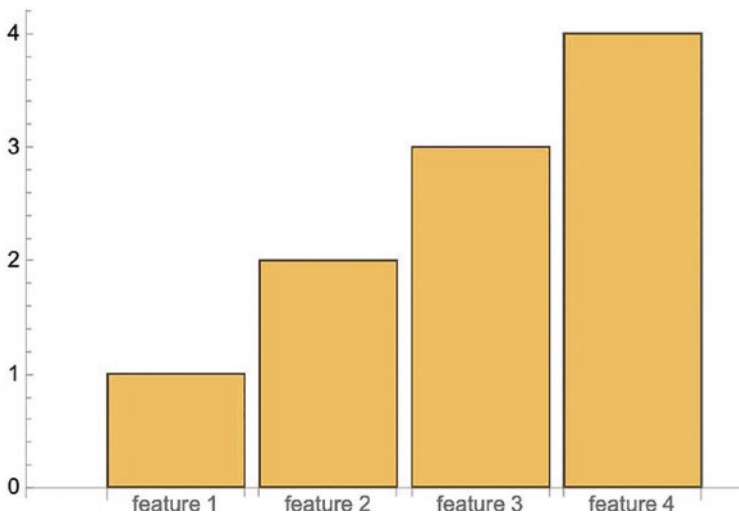
## Statistical Charts

Using charts to display data is a straightforward approach with Mathematica. Many times, studies include various types of information. Mathematica has a repertoire of statistical charts based on users' needs for more visual and understandable presentations.

### Bar Charts

Sometimes, when you conduct a statistical study, you can find quantitative and qualitative variables and create a bar graph representation for these variables. A bar graph (see Figure 6-1) is a graphical representation where the number of frequencies of a discrete qualitative variable is displayed on an axis.

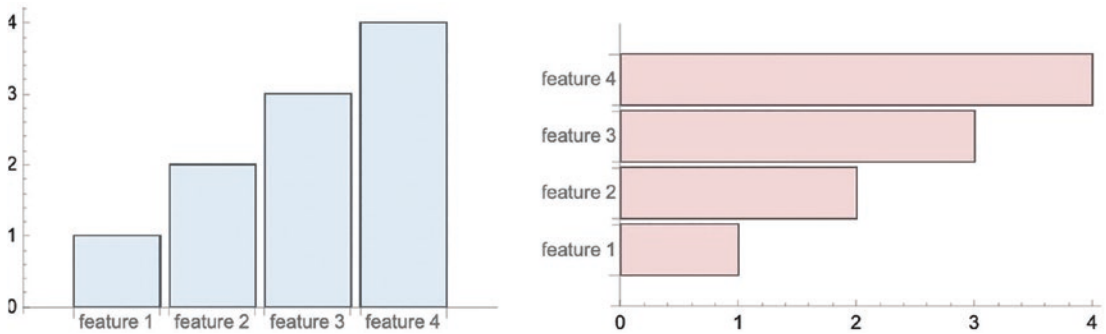
```
In[44]:= BarChart[{1,2,3,4},ChartLabels->{"feature 1","feature 2",  
"feature 3","feature 4"}]  
Out[44]=
```



**Figure 6-1.** Bar chart

The different modalities of the qualitative variable are positioned on one of the axes. The other axis shows the value or frequency of each category on a given scale. The feature 2 bar has an associated value of 2. The orientation of the graph can be vertical, where the categories are located on the horizontal axis, and the bars are vertical or horizontal, where the categories are located on the vertical axis. The bars are horizontal (see Figure 6-2).

```
In[45]:= GraphicsRow[{BarChart[{1,2,3,4},ChartLabels->{"feature 1",
"feature 2","feature 3","feature 4"},BarOrigin->Bottom,ChartStyle->
LightBlue],BarChart[{1,2,3,4},ChartLabels->{"feature 1","feature
2","feature 3","feature 4"},BarOrigin->Left,ChartStyle->LightRed]}]
Out[45]=
```



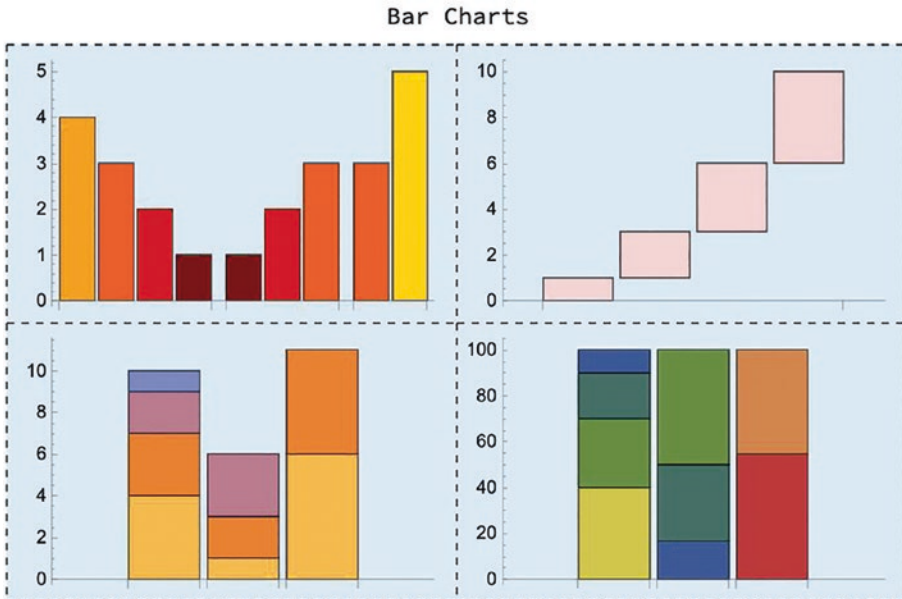
**Figure 6-2.** Bottom and left origin bar chart

Bar graphs can be used to compare magnitudes of different categories and observe how values change according to a fixed variable—for example, each feature. In addition, you can choose how to show the bars, where you show a single series, as shown in the earlier example; grouped, which contains several data series and is represented by a different type of bar; or stacked, where the bar is divided into segments with different colors representing various categories. The percentile layout is displayed on a percentage scale, as shown in Figure 6-3.

```
In[46]:= Labeled[GraphicsGrid[{{BarChart[{{4, 3, 2, 1}, {1, 2, 3}, {3, 5}},
ChartLayout -> "Grouped", ColorFunction -> "SolarColors"], BarChart[{1, 2,
3, 4}, ChartStyle -> LightRed, ChartLayout ->
"Stepped"]}, {BarChart[{{4, 3, 2, 1}, {1, 2, 3}, {6, 5}}, ChartLayout ->
"Stacked"], BarChart[{{4, 3, 2, 1}, {1, 2, 3}, {6, 5}}, ChartLayout ->
```



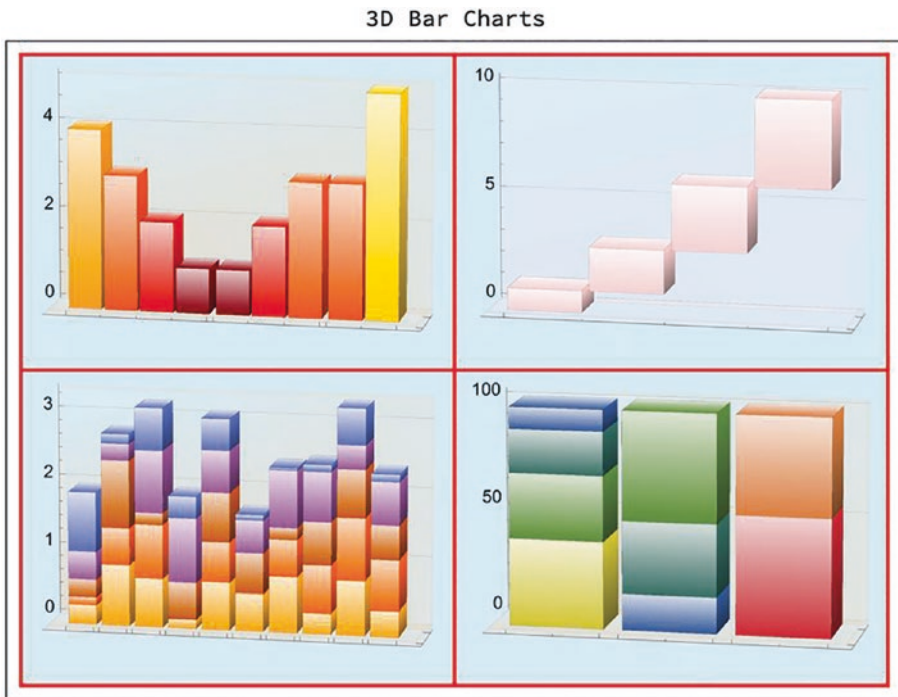
```
"Percentile", ColorFunction -> "DarkRainbow"]}},
Frame -> All, FrameStyle -> Directive[Black, Dashed], Background ->
LightBlue, ImageSize -> 500], "Bar Charts", Top]
Out[46]=
```



**Figure 6-3.** Bar chart grid

There is also the counterpart to 3D graphics, with `BarChart3D` (see Figure 6-4).

```
In[47]:= SeedRandom[123];
Labeled[GraphicsGrid[{{BarChart3D[{{4, 3, 2, 1}, {1, 2, 3}, {3, 5}},
ChartLayout -> "Grouped", ColorFunction -> "SolarColors"],
BarChart3D[{{1, 2, 3, 4}, ChartStyle -> LightRed, ChartLayout ->
"Stepped"]}, {BarChart3D[RandomReal[1, {10, 5}], ChartLayout -> "Stacked"],
BarChart3D[{{4, 3, 2, 1}, {1, 2, 3}, {6, 5}], ChartLayout -> "Percentile",
ColorFunction -> "DarkRainbow"]}}, Frame -> All, FrameStyle ->
Directive[Red, Thick], Background -> LightBlue, ImageSize -> 500], "3D Bar
Charts", Top, Frame -> True, Background -> White]
Out[48]=
```

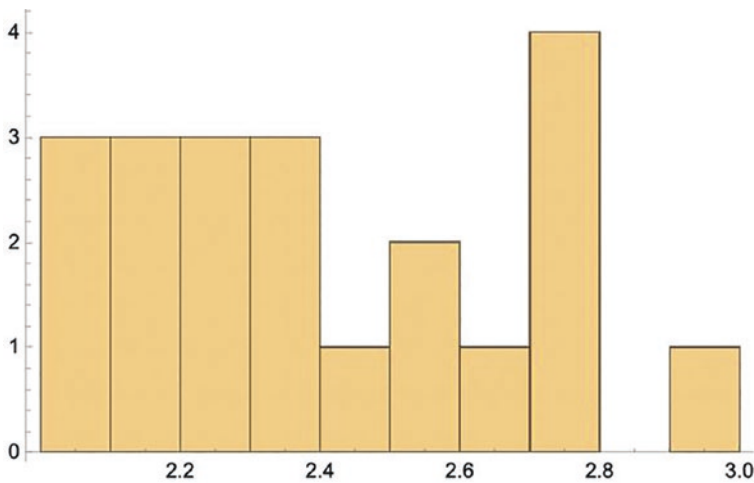


**Figure 6-4.** 3D bar charts grid

## Histograms

Histograms are a type of visualization that is commonly used in statistical studies. With histograms, you can see how a sample is distributed. Histograms are used to represent the frequencies of a quantitative variable. The variable classes are positioned on the horizontal axis, and the frequencies are on the other axis. The following examples graph a histogram from a population of 50 random values between 0 and 1 and set the number of bins to 10. The second argument for histograms is to define the number of bins (see Figure 6-5).

```
In[49]:= SeedRandom[4322];
hist1=Table[RandomReal[{2,3}],{i,0,20}];
Histogram[hist1,10]
Out[51]=
```



**Figure 6-5.** Histogram for random real numbers

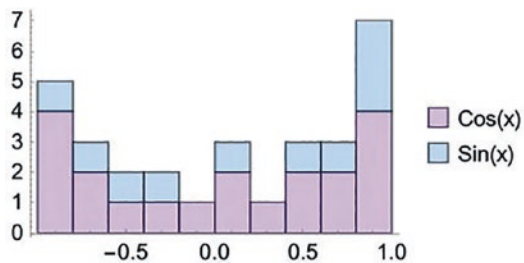
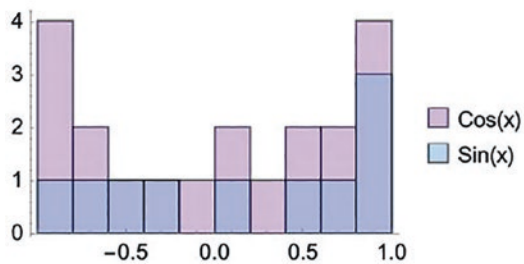
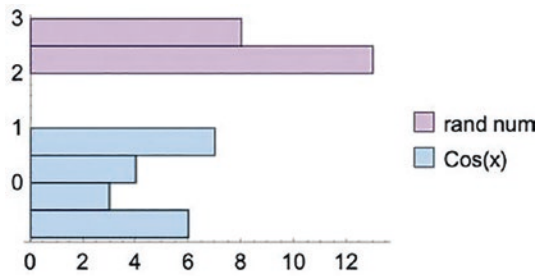
---

**Note** When dealing with charts, if you put the pointer cursor on the graphic, an info tip marks the value.

---

Just like with bar charts, there are ways to edit the histogram's origin and how the histogram is displayed—stacked or overlapped—as shown in Figure 6-6.

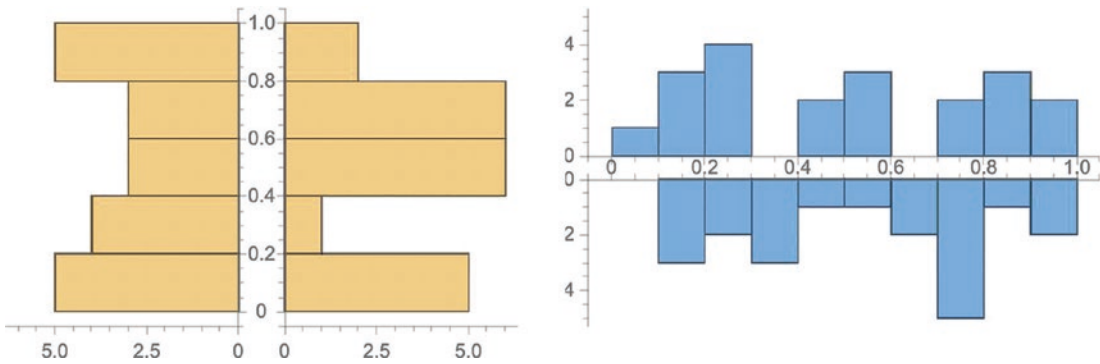
```
In[51]:= hist2=Table[Cos[i],{i,1,20}];
hist3=Table[Sin[i],{i,1,10}];
GraphicsColumn[{Histogram[{hist1,hist2},10,BarOrigin->
Left,ChartStyle->"Pastel",ChartLegends->{"rand num",
"Cos(x)"},Histogram[{hist2,hist3},10,ChartLayout->
"Overlapped",ChartStyle->"Pastel",ChartLegends-> {"Cos(x)","Sin(x)"}},
Histogram[{hist2,hist3},10,ChartLayout-> "Stacked",ChartStyle->
"Pastel",ChartLegends->{"Cos(x)","Sin(x)"}]]}]]
Out[54]=
```



**Figure 6-6.** Histogram shapes grid

With this in mind, you can also graph bidirectional histograms using `PairedHistograms`. These can be horizontal or vertical orientations and contain two data series whose bars go opposite directions (see Figure 6-7).

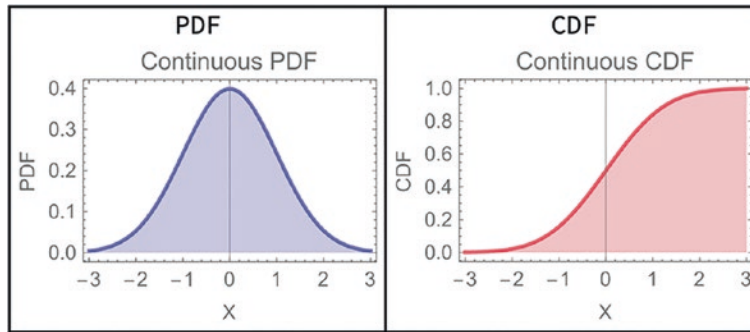
```
In[55]:=SeedRandom[123] ;GraphicsRow[{PairedHistogram[{RandomReal[{0,1},20]},
{RandomReal[{0,1},20]},BarOrigin->Left], PairedHistogram[{RandomReal
[{0,1},20]}, {RandomReal[{0,1},20]},10,BarOrigin->Top, ChartStyle->"Pastel"]}]]
Out[55]=
```



**Figure 6-7.** Paired histograms with different origins

While histograms offer a powerful way to visualize data distribution, you can enhance these visualizations by incorporating various statistical functions directly into the notebook. By default, histograms in the Wolfram Language display data counts within each bin. However, it's often valuable to visualize cumulative distribution functions (CDFs) and probability density functions (PDFs), like the following example and Figure 6-8.

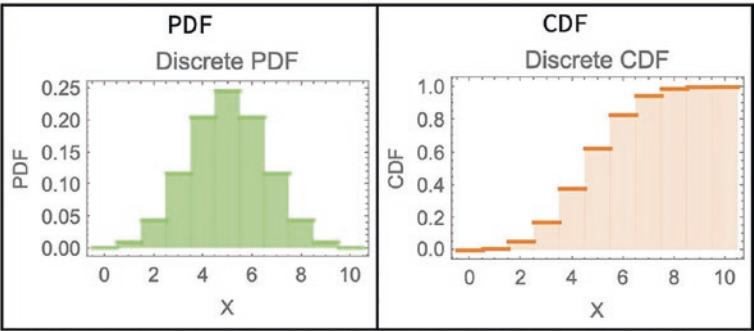
```
In[56]:= (*common options*)
continuousOpts = {Filling -> Axis, Frame -> True, FrameLabel -> {"X", #},
PlotLabel -> "Continuous " <> #} &;
(*Continuous PDF and CDF plots*)
continuousPlots =
  Grid[{{Labeled[Plot[PDF[NormalDistribution[0, 1], x], {x, -3, 3},
Evaluate@continuousOpts["PDF"], PlotStyle -> Directive[Blue,
Opacity[0.5]]], "PDF", Top], Labeled[Plot[CDF[NormalDistribution[0,
1], x], {x, -5, 5}, Evaluate@continuousOpts["CDF"], PlotStyle ->
Directive[Red, Thick]], "CDF", Top]}}, Frame -> All]
Out[57]=
```



**Figure 6-8.** PDF and CDF plots for the standard normal distribution

The previous code generates the PDF and CDF plots for a continuous distribution, for a standard normal distribution (mean 0, standard deviation 1). It uses the distributions as arguments for the PDF and CDF functions. The plots are labeled accordingly for clarity. Similarly, the process can be done to discrete distributions (see Figure 6-9).

```
In[57]:= (*common options*)
discreteOpts = {ExtentSize -> Full, Frame -> True, FrameLabel -> {"x", #},
PlotLabel -> "Discrete " <> #} &;
(*Discrete PDF and CDF plots*)
discretePlots = Grid[{{Labeled[ DiscretePlot[PDF[BinomialDistribution
[10, 0.5], x], {x, 0, 10}, Evaluate@discreteOpts["PDF"], PlotStyle ->
Directive[Green, Opacity[0.5]]], "PDF", Top], Labeled[DiscretePlot[CDF
[BinomialDistribution[10, 0.5], x], {x, 0, 10}, Evaluate@discreteOpts["CDF"],
PlotStyle -> Directive[Orange, Thick]], "CDF", Top]}}, Frame -> All]
Out[58]=
```

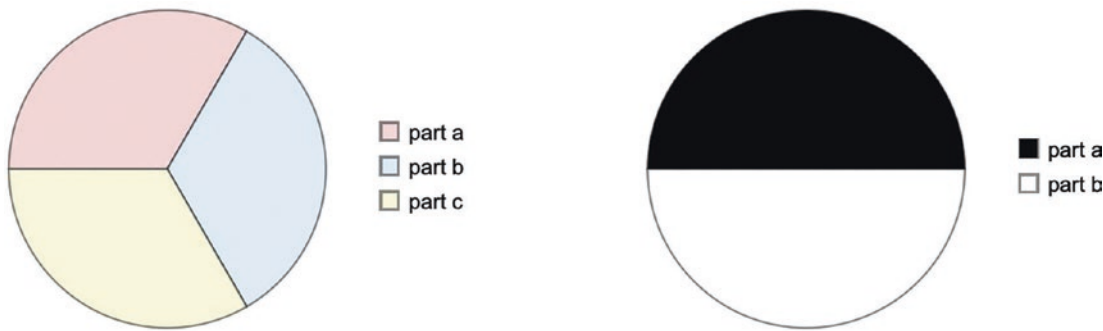


**Figure 6-9.** PDF and CDF plots for the binomial distribution with parameters  $n=10$  and  $p=0.5$

### Pie Charts and Sector Charts

Pie charts are circles that are divided into two or more sections. They represent quantitative variables that make up a total; for example, the sector's size is drawn proportional to the value it represents and is expressed in percentages, which only provides relative quantitative information. Pie charts are made with the `PieChart` command (see Figure 6-10).

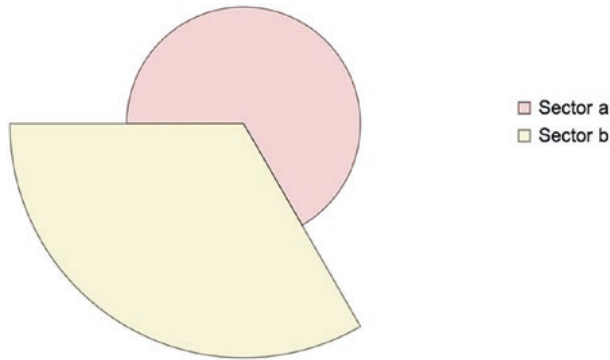
```
In[59]:= GraphicsRow[{PieChart[{1,1,1},ChartLegends->{"part a","part b",
"part c"},ChartStyle->{LightRed,LightBlue,LightYellow}], PieChart[{1,1},
ChartLegends->{"part a","part b"},ChartStyle-> "SunsetColors"]}]]
Out[59]=
```



**Figure 6-10.** Pie charts

Sector charts are graphed with the `SectorChart` command (see Figure 6-11). They are used to compare different data that occur in the same place. They are constructed from the proportional size of  $x$  to the value of the radius of  $y$ . The dimension in which the quantities are expressed must be the same for all the segments.

```
In[60]:= SectorChart[{{2,1},{1,2}},ChartLegends->{"Sector a","Sector b"},
ChartStyle->{LightRed,LightYellow}]
Out[60]=
```

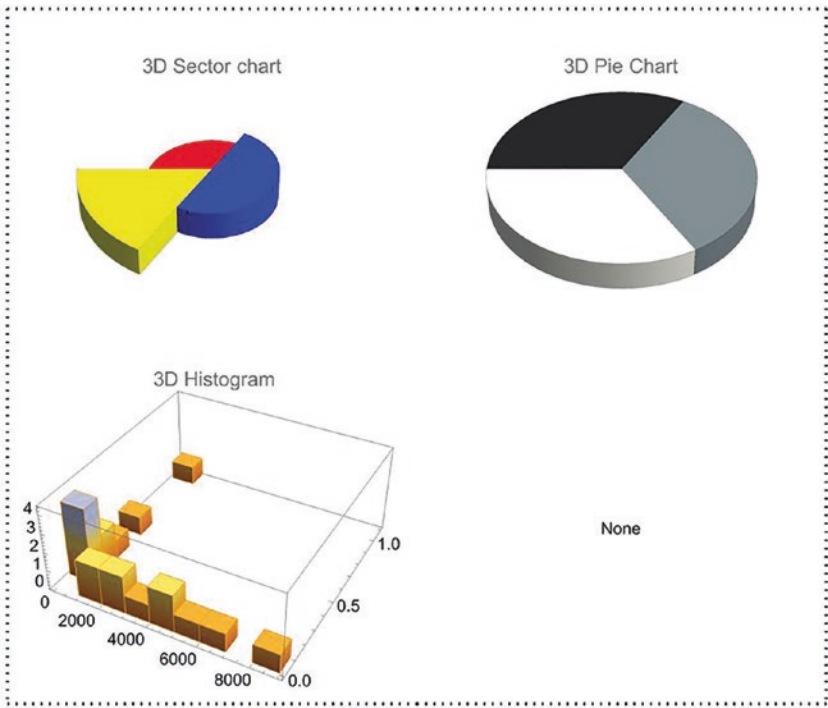


**Figure 6-11.** Sector chart

For each graph seen, there is a corresponding command to create them in 3D, as shown in Figure 6-12.

```
In[61]:=
GraphicsGrid[{{SectorChart3D[{{2, 1, 1}, {3, 1, 2}, {1, 2, 2}},
PlotLabel -> "3D Sector chart", ChartStyle -> {Red, Blue,
Yellow}], PieChart3D[1, 1, 1, ChartStyle -> "GrayTones", PlotLabel
-> "3D Pie Chart"]}, {Histogram3D[ Table[{i^3, i^-1}, {i, 20}], 10,
ChartElementFunction -> "GradientScaleCube", PlotLabel -> "3D
Histogram"], None}}, ImageSize -> 500, Frame -> True, FrameStyle ->
Directive[Thick, Dotted]]
Out[61]=
```



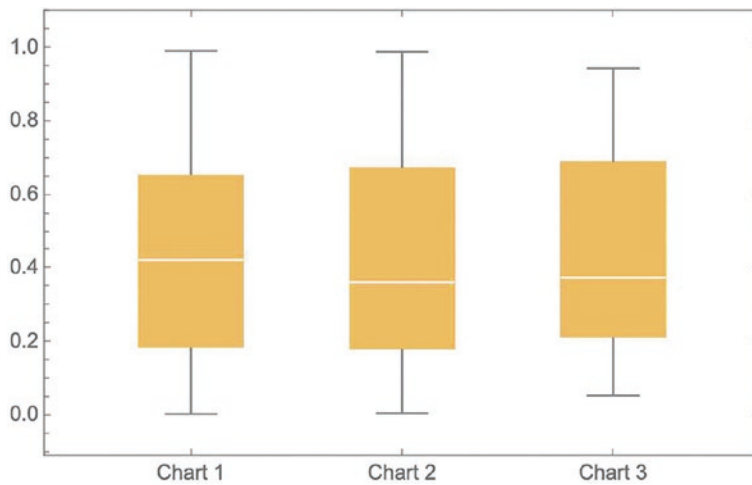


**Figure 6-12.** 3D grid charts

## Box Plots

The box plot is a way of representing and observing a data distribution. Fundamentally, it highlights aspects of data distribution in one or more series. To graph a box plot, you use the `BoxWhiskerChart` command (see Figure 6-13).

```
In[62]:= SeedRandom[1234] BoxWhiskerChart[{Table[RandomReal[],{i,0,50}],
Table[RandomReal[],{i,0,50}], Table[RandomReal[],{i,0,15}]},ChartLabels→
{"Chart 1","Chart 2","Chart 3"}]
Out[62]=
```

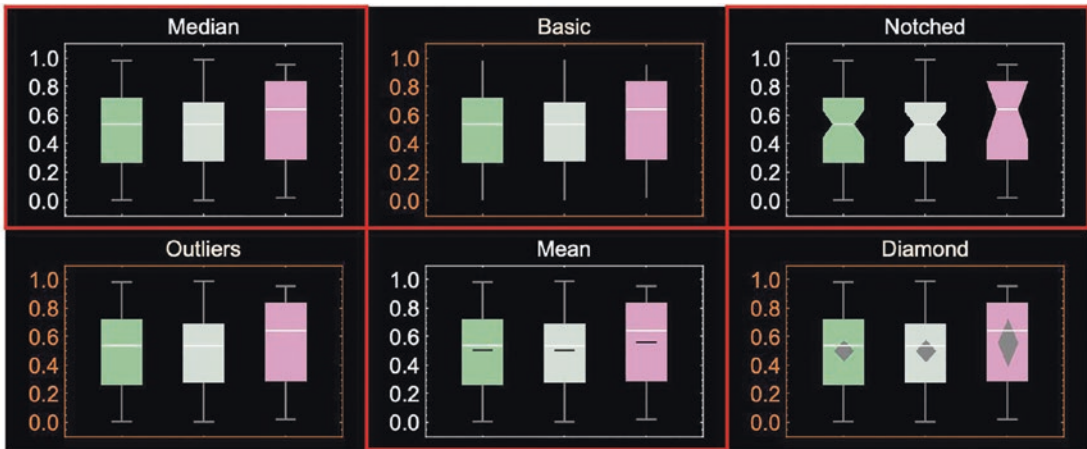


**Figure 6-13.** *Box plot*

The box is represented by a rectangle that marks the interquartile range of the distribution. The first line from bottom to top marks the value of the first quartile (25%), the line that crosses the box is the median, and the last line that delimits the box is the third quartile (75%). Whiskers are the lines that mark the maximum and minimum values. When passing the mouse cursor over the plot, information about the data is shown; this includes minimum, maximum, median, 75th percentile, and first quartile. Depending on the specification, this can affect the parameters displayed and how (see Figure 6-14).

```
In[62]:= SeedRandom[123];
data = {Table[RandomReal[], {i,0,50}],Table[RandomReal[], {i,0,50}],
Table[RandomReal[], {i,0,15}]}];
options = {ImageSize -> Medium, ChartStyle -> "MintColors", FrameStyle ->
Directive[White, 12]};
GraphicsGrid[{{BoxWhiskerChart[data, "Median", PlotLabel -> Style["Median",
White], options], BoxWhiskerChart[data, "Basic", PlotLabel ->
Style["Basic", LightOrange], FrameStyle -> Directive[Orange, 12], options],
BoxWhiskerChart[data, "Notched",
PlotLabel -> Style["Notched", White], options]},
{BoxWhiskerChart[data, "Outliers", PlotLabel -> Style["Outliers",
LightOrange], FrameStyle -> Directive[Orange, 12], options],
BoxWhiskerChart[data, "Mean", PlotLabel -> Style["Mean",
```

```
White], options],BoxWhiskerChart[data, "Diamond", PlotLabel ->
Style["Diamond", LightOrange], FrameStyle -> Directive[Orange, 12],
options]]},FrameTicksStyle -> 18, Frame -> {None, None, {{1, 1} -> True,
{2, 2} -> True, {1, 3} -> True}}, FrameStyle -> Directive[Thick, Red],
Background -> Black]
Out[63]=
```



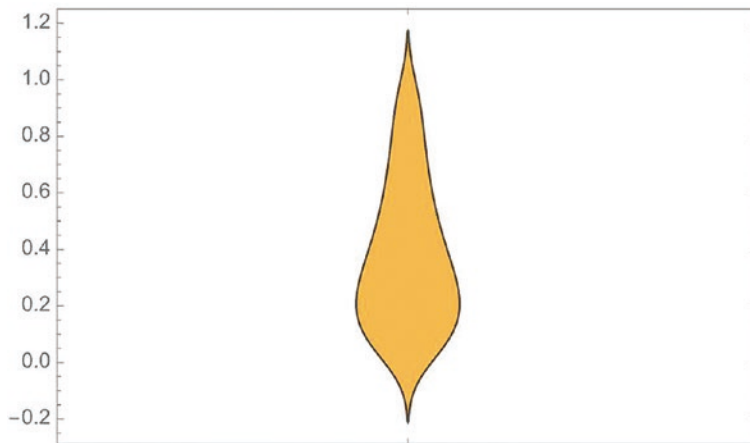
**Figure 6-14.** Multiple box plots

Median is the default specification; it shows the median in the center of the box. Basic is to show only the box. Notches show the confidence interval for the median. Outliers show and mark the atypical points. The mean marks the average of the distribution, and Diamond notes the confidence interval for the mean.

## Distribution Chart

A violin diagram is used to visualize the distribution of the data and the probability density. To plot a violin plot (see Figure 6-15), the `DistributionChart` command is used.

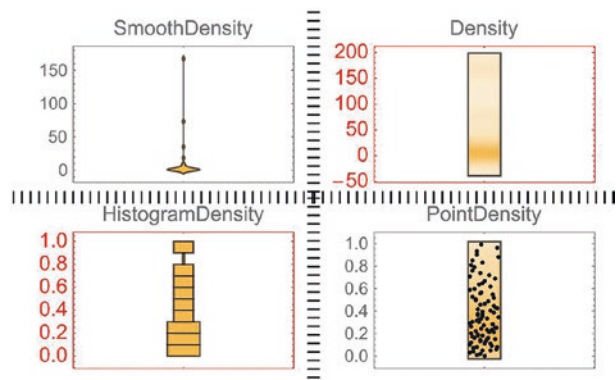
```
In[64]:= DistributionChart[Table[i^Exp[i],{i,0,1,0.01}]]
Out[64]=
```



**Figure 6-15.** *Violin plot*

The graph in the figure combines a box-and-whisker plot and a density plot on each side to show how the data is distributed. `DistributionChart` has different shapes to graph (see Figure 6-16).

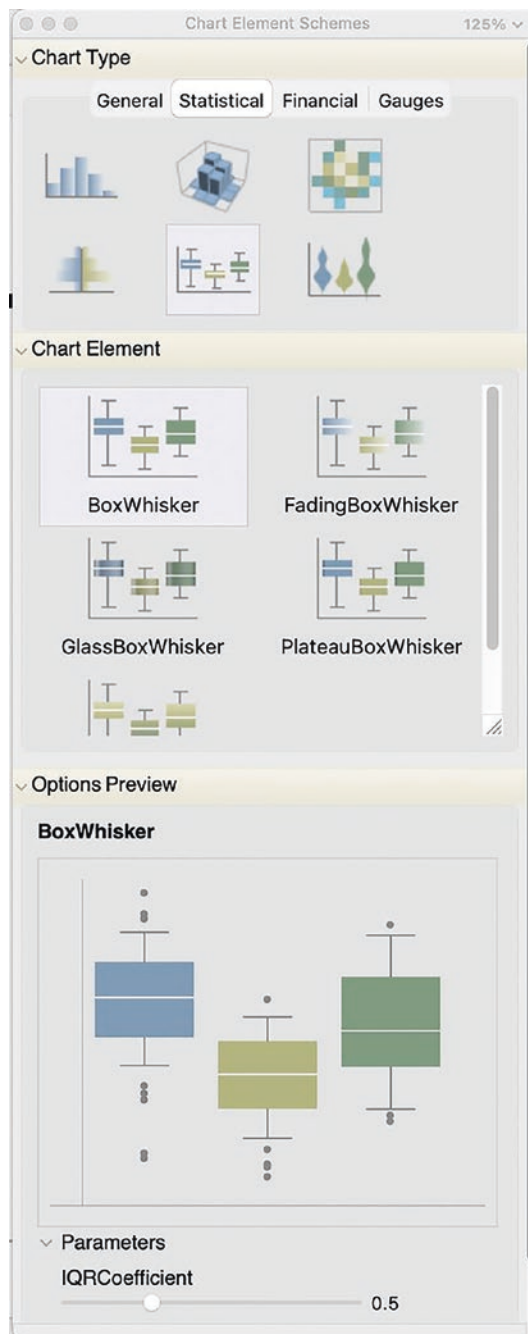
```
In[65]:= GraphicsGrid[{{DistributionChart[Table[i^Exp[i], {i, 0, 2, 0.1}],
ChartElementFunction -> "SmoothDensity", PlotLabel -> "SmoothDensity"],
DistributionChart[Table[i^Exp[i], {i, 1, 2, 0.1}], ChartElementFunction ->
"Density", PlotLabel -> "Density",
FrameStyle -> Directive[Red, 12]]}, {DistributionChart[ Table[i^Exp[i],
{i, 0, 1, 0.09}], ChartElementFunction -> "HistogramDensity", PlotLabel ->
"HistogramDensity", FrameStyle -> Directive[Red, 12]], DistributionChart
[Table[i^Exp[i], {i, 0, 1, 0.0112}], ChartElementFunction -> "PointDensity",
PlotLabel -> "PointDensity"]}}, ImageSize -> Medium, FrameStyle ->
Directive[Thickness[0.02], LightGray], Dividers -> {2 -> Directive[Black,
Dotted], 2 -> Directive[Black, Dotted]}, Frame -> {1 -> False, False}]
Out[65]=
```



**Figure 6-16.** Violin plots in different shapes

## Charts Palette

Another way to add options to charts is through the Chart Element Schemes palette, found within the Palettes menu (Palettes ► Chart Element Schemes). This palette is shown in Figure 6-17.

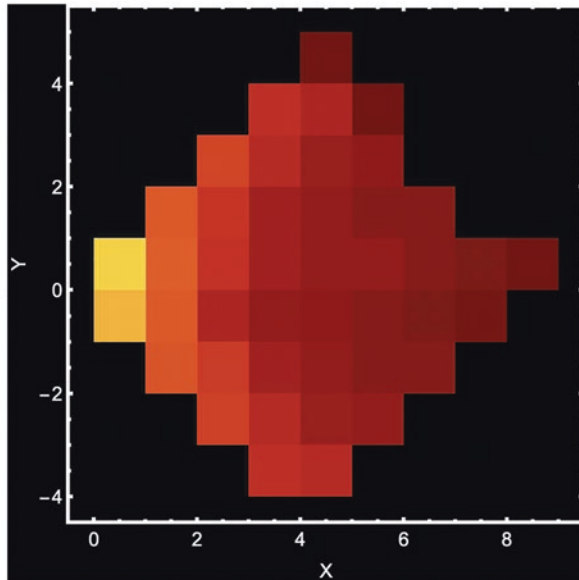


**Figure 6-17.** Chart Element Schemes palette

In the palette, you find three categories. Chart Type is where you choose the type of chart. This contains four tabs: (1) general, where the graphics are found from bar charts, sector, footer, and others; (2) statistical graphs associated with data distributions; (3) financial, associated with charts for financial data; and (4) gauges, which are diagrams of measures. The second category is to choose the shape of the graph with the `ChartElementFunction` option. The third category is for the preview of the options chosen from the previous categories.

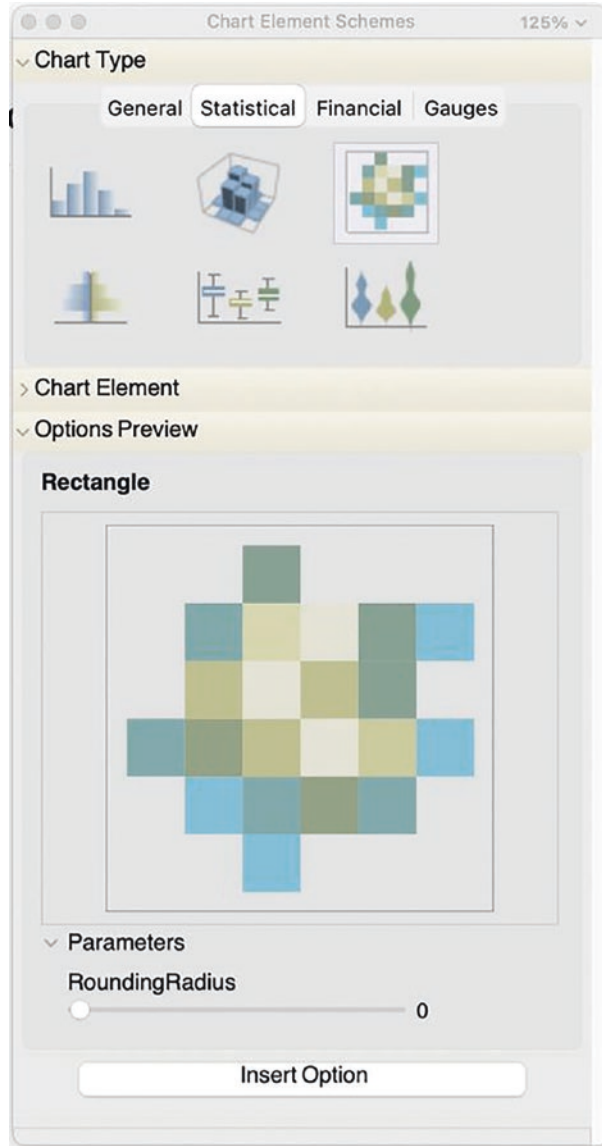
To illustrate this, let's look at the following exercises. First, make the graph of the density of a histogram, and later, modify the shape of the graph with the help of the palette. To graph the density of a histogram, use the `DensityHistogram` command (see Figure 6-18).

```
In[66]:= DensityHistogram[Flatten[Table[{x^2+y^2,x^2-y^2},
{x,0,2,0.1},{y,0,2,0.1}],1],ChartBaseStyle->Red,ColorFunction->
"SolarColors",Background->Black,FrameStyle->Directive[White,Thick],
FrameLabel->{"X","Y"},ImageSize->300]
Out[66]=
```



**Figure 6-18.** Density histogram

Once the graph is done, add an option with the pallet head and open the Chart Element Schemes palette. Within the chart type, you click the statistical tab and choose the DensityHistogram chart. Once the chart has been selected, go to Chart Element and select that the type of form is Bubble. Then go to Options Preview to see how the graph would look; if you click Shape, a pop-up menu appears with other shapes; you choose hexagon. Figure 6-19 shows how the preview of the selected chart elements should look.

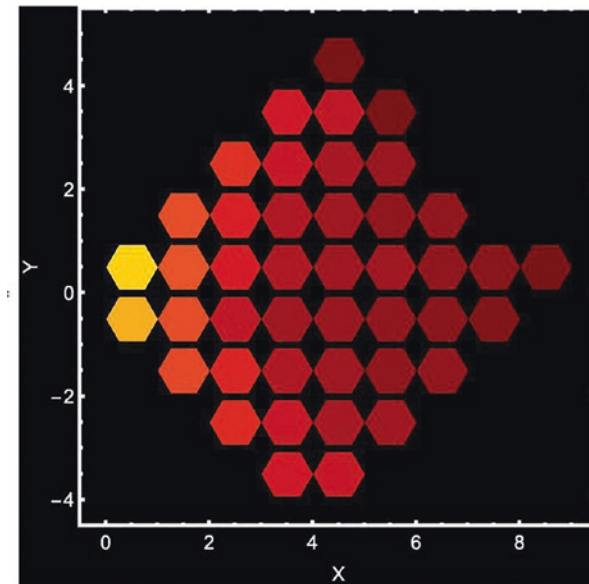


**Figure 6-19.** Density histogram options selected



Once you finish selecting, click the insert button so that it inserts the following code: `ChartElementFunction ▶ ChartElementDataFunction ["Bubble", "Shape" ▶ "Hexagon"]`. To graph it correctly, add this code as an option and proceed to plot it (see Figure 6-20) to observe the new option added.

```
In[67]:= DensityHistogram[Flatten[Table[{x^2 + y^2, x^2 - y^2}, {x, 0,
2, 0.1}, {y, 0, 2, 0.1}], 1], ChartBaseStyle -> Red, ColorFunction ->
"SolarColors", Background -> Black, FrameStyle -> Directive[White,
Thick], FrameLabel -> {"X", "Y"}, ImageSize -> 300, ChartElementFunction ->
ChartElementDataFunction["Bubble", "Shape" -> "Hexagon"]]
Out[67]=
```



**Figure 6-20.** *Hexagon density histogram*

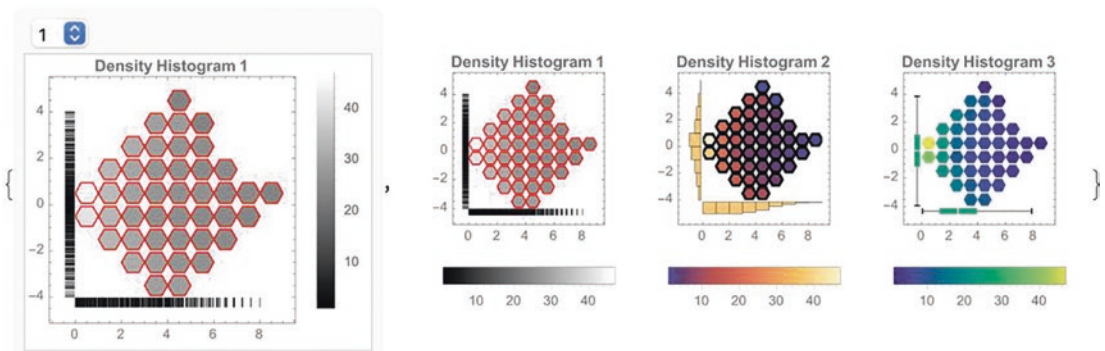
The `DensityHistogram` command allows you to choose how to display the data distribution along the axes; it can be the dimensions, box plots, or histograms if you select the `Method` type as an option (see Figure 6-21).

```
In[68]:= hist = Flatten[Table[{x^2+y^2,x^2-y^2}, {x,0,2,0.1},
{y,0,2,0.1}],1];
densityHistogram[distAxes_, colFunc_, baseStyle_, plotLabel_,
imgSize_] := DensityHistogram[Hist, Method -> {"DistributionAxes" ->
```

```

distAxes},ColorFunction -> colFunc, ChartBaseStyle -> baseStyle, PlotLabel
-> Style[plotLabel, Bold], ChartLegends -> Automatic,
ChartElementFunction -> ChartElementDataFunction["Bubble", "Shape"
-> "Hexagon"],ImageSize -> imgSize] {MenuView[{densityHistogram
[True, GrayLevel, Directive[FaceForm[Opacity[0.5]], EdgeForm[Red]],
"Density Histogram 1", 200], densityHistogram["Histogram",
Automatic, Directive[EdgeForm[Thick]], "Density Histogram 2", 200],
densityHistogram["BoxWhisker", "BlueGreenYellow", Automatic, "Density
Histogram 3", 200]}], GraphicsRow[{densityHistogram[True, GrayLevel, Direct
ive[FaceForm[Opacity[0.5]], EdgeForm[Red]],
"Density Histogram 1", 130], densityHistogram["Histogram", Automatic,
Directive[EdgeForm[Thick]], "Density Histogram 2", 130],
densityHistogram["BoxWhisker", "BlueGreenYellow", Automatic, "Density
Histogram 3", 130]}]}
Out[70]=

```



**Figure 6-21.** Menu view of the three different method plots

The plots are shown inside as a menu, so to access the different graphs, you have to select each graph within the menu. Even so, you show the plots on a small scale to demonstrate how they should look (see Figure 6-21). The first graph shows the dimensions of the data distribution along the axes. The second shows the distribution of the data in the form of histograms, and the third shows the box plots.

# Ordinary Least Squares Method

The ordinary least squares method finds the best-fitting line through data points. This method is used to study the relationship between the dependent variable and the independent variable. The process is based on the expression of finding a line of the form  $y = mx + b$ , where  $x$  is the independent variable,  $y$  is the dependent variable,  $m$  is the slope, and  $b$  is the y-intercept. The slope and the sorted to origin  $b$  are calculated from the following equations.

$$m = \frac{n * \sum (x * y) - \sum x * \sum y}{n * \sum x^2 - |\sum x|^2}$$

$$b = \frac{\sum y * \sum x^2 - \sum x * \sum (x * y)}{n * \sum x^2 - |\sum x|^2}$$

The summation is denoted by the Greek capital letter sigma ( $\sum$ );  $n$  is the amount of data in the sample. The method is calculated for measured data pairs and slope values, and y-intercept sources are calculated to create the best data fit to a line. By substituting in the general equation, you get the equation of the line for the dataset.

To illustrate the method, let's look at the following example using the points for the dependent and the independent variables.

```
In[71]:= data={{-1,10},{0,9},{1,7},{2,5},{3,4},{4,3},{5,0},{7,-1}};
Grid[Transpose[Prepend[data,{"X","Y"}]],Dividers->{2->True,2->
True},Alignment-> Center]
Out[72]=
```

```
X | -1 0 1 2 3 4 5 7
___|_____
Y | 10 9 7 5 4 3 0 -1
```

Next, calculate the data needed to get the slope and y-intercept.

```
In[73]:=n = Length[data];
sumX = Total@data[[All, 1]];
sumY = Total@data[[All, 2]];
sumXY = Total[data[[All, 1]]*data[[All, 2]]];
sumXSqr = Total@(data[[All, 1]]^2);
```

```

m = N@((n*sumXY-sumX*sumY)/n*sumXSqr-Abs[sumX]^2);
b = N@((sumY*sumXSqr-sumX*sumXY)/n*sumXSqr-Abs[sumX]^2);

```

Use the Solve command to solve the equation of the shape  $y = mx + b$ . The first argument is the equation, and the second is for the variable to solve. You must use the same double notation to enter the equation since a single equal is for set instruction.

```

In[80]:= Solve[SetPrecision[y==m*x+b,3],y]
Out[80]= {{y->8.47-1.47 x}}

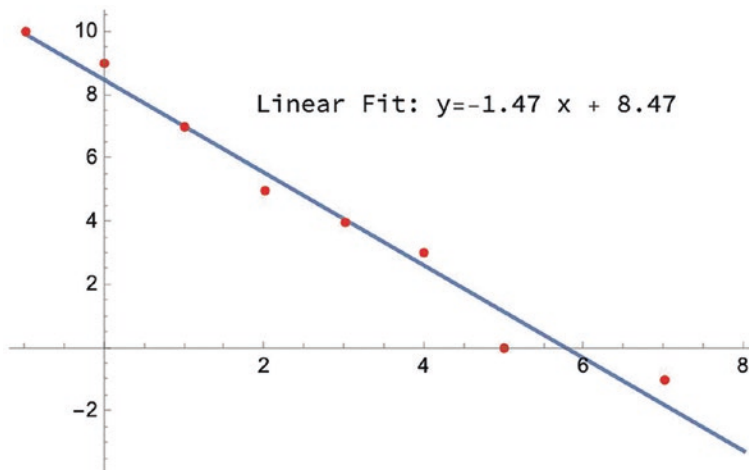
```

This results in the equation of the line being  $y = 1.47x + 8.47$ . Given this equation, you plot the points and the line that best fits these points (see Figure 6-22).

```

In[81]:= Show[Plot[b + m*x,{x,-1,8}, PlotLegends->Placed["Linear Fit: y=-
1.47x+8.47",{0.6,0.8}],PlotRange->Automatic], ListPlot[data, PlotStyle
-> Red]]
Out[81]=

```



**Figure 6-22.** Plot of data and fitted curve

Having obtained the equation, you observe that this is a model with a negative slope, corroborated by the equation graph shown in blue.

## Pearson Coefficient

The measure that tells you that both the points fit the equation is the Pearson correlation coefficient named  $r$ . When the points are found with a positive slope,  $r$  has a positive value. When the points are negatively sloped,  $r$  has a negative value. The coefficient value determines the correct setting, ranging from  $-1$  to  $1$ . When the  $r$  value is  $1$  or  $-1$ , it tells you that the points are adjusted exactly to the line. The closer  $r$  is to  $-1$  or  $1$  indicates that there appears to be a linear relationship between the study variables. Otherwise, when  $r$  is equal to  $0$ , it tells you that the setting is not correct, and therefore, it can be concluded that there is no apparent linear relationship.

The equation for determining the coefficient is as follows.

$$r = \frac{\text{Cov}(x * y)}{\sigma_x \sigma_y},$$

Cov represents the covariance of  $x$ ,  $y$ . The symbols  $\sigma x$  and  $\sigma y$  represent the standard deviations of  $x$  and  $y$ .

Now, you proceed to calculate the coefficient  $r$  for the created adjustment. For this, you must introduce only the points of  $x$  and  $y$ , for calculating covariance and standard deviations.

```
In[82]:= r= N@(Covariance@@{data [[All,1]],data [[All,2]]} /
(StandardDeviation@data[[All,1]]*StandardDeviation@data[[All,2]]))
Out[82]= -0.987814
```

The result is close to  $1$ ; therefore, the straight is adequately fair to the data. Although it is possible to calculate it through the equation, Mathematica has a function for this calculation. Correlation calculates the coefficient from two lists, so you need to enter only the  $x$  data in one list and the data from  $y$  in another list.

```
In[83]:= N@Correlation[data[[All,1]],data[[All,2]]]
Out[83]= -0.987814
```

And you get the same result as the previous one.

## Linear Fit

Mathematica has functions that specialize in finding the best linear model using `LinearModelFit`. Given the dataset, you write the `LinearModelFit` command with the data to work and the variable to write the equation. In addition, you can specify the level of precision for adjustment with `WorkingPrecision`.

```
In[84]:= model=LinearModelFit[data,x,x,WorkingPrecision->10]
Out[84]= FittedModel[8.473684211-1.466165414 x]
```

The same equation returns to you but with better precision. Within the model, you can access different properties related to the data, the model, and other adjustment parameters, as well as measures of the goodness of the fit, among others. To illustrate this, you see how to do it for the `BestFit`, `BestFitParameters`, and `Function` options, which return the best-fit equation as a list, the best parameters, and model construction for a pure function, respectively.

A critical aspect is that trying to make predictions about a future value using the fitted equation ( $8.47 - 1.47x$ ), with values of  $x$  outside the range, could generate abnormal values since you have not established whether the relation of the equation outside the range of  $x$  is met. Figure 6-23 shows the fitted curve calculations.

```
In[85]:= {"\n" Framed["Best Fit Parameters b and m: " <>
ToString[model["BestFitParameters"]], Background -> LightYellow], "\n"
Framed["Equation: " <> ToString[model["BestFit"]], Background ->
LightYellow], "\n" Framed["Pure Function:" <> ToString[SetPrecision[model[
"Function"], 3]], Background -> LightYellow], "\n" Framed["r coeficcient:"
<> ToString[r], Background -> LightYellow]}
Out[85]=
```

```
{
  Best Fit Parameters b and m: {8.473684211, -1.466165414},
  Equation: 8.473684211 - 1.466165414 x,
  Pure Function:8.47 - 1.47 #1 & ,
  r coeficcient:-0.987814 }
```

**Figure 6-23.** Fitted parameters, equation, and Pearson coefficient

Since you have the line that best fits, you should consider whether a relationship exists between  $x$  and  $y$ . How do you know if the adjustment adequately describes the linear relationship between the  $x$  and  $y$  variables? To solve this problem, there is the concept of residual.

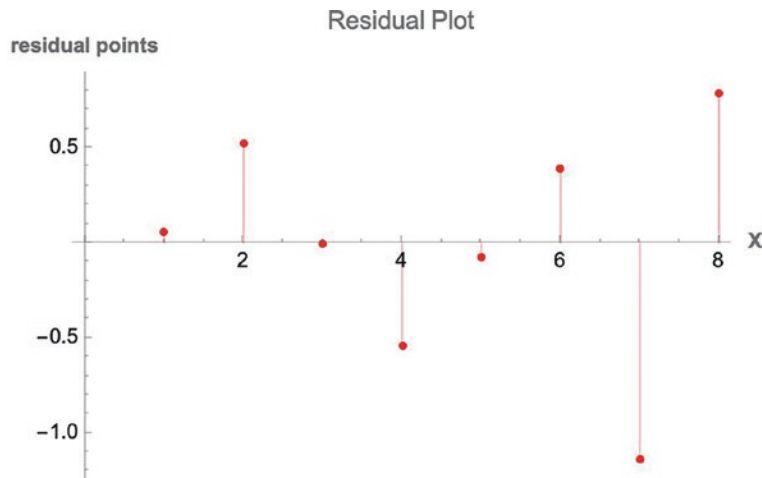
## Model Properties

Residuals can be used as a measure to know how good the fit of the line is to the study points. Residuals are vertical deviations, either positive or negative. A residual point is the difference between the observed value of the dependent variable and the value that predicts the adjustment. To get the residual points, write the `FitResiduals` property within the model.

```
In[86]:= model["FitResiduals"]
Out[86]= {0.06015038,0.52631579,-0.00751880,-0.54135338,-0.07518797,
0.39097744,-1.14285714,0.78947368}
```

With these points, you can get the residual plot (see Figure 6-24), which is the  $x$  variable vs. the residual points.

```
In[87]:= ListPlot[model["FitResiduals"],PlotStyle->{Red,Thick},
PlotLabel->"Residual Plot",AxesLabel-> {Style["X",Bold], Style["residual
points",Bold]},Filling->Axis]
Out[87]=
```



**Figure 6-24.** Residual plot of the fitted data

To show only the observed and predicted values for the single prediction, use the `SinglePredictionConfidenceIntervalTable` option.

```
In[88]:= model["SinglePredictionConfidenceIntervalTable"]
```

```
Out[88]=
```

Observed	Predicted	Standard Error	Confidence Interval
10	9.93984962	0.78481739	{8.0194706, 11.8602286}
9	8.47368421	0.74856412	{6.6420138, 10.3053546}
7	7.00751880	0.72287410	{5.2387096, 8.7763280}
5	5.54135338	0.70889670	{3.8067456, 7.2759611}
4	4.07518797	0.70732661	{2.3444221, 5.8059538}
3	2.60902256	0.71824519	{0.8515399, 4.3665052}
0	1.14285714	0.74110068	{-0.6705509, 2.9562652}
-1	-1.78947368	0.81811053	{-3.7913180, 0.2123707}

In addition to the residual points, you can extract the table from the parameters of the model adjusted with the `ParameterTable` property.

```
In[89]:= model["ParameterTable"]
```

```
Out[89]=
```



	Estimate	Standard Error	t-Statistic	P-Value
1	8.473684211	0.34167121	24.800697	$2.8278226 \cdot 10^{-7}$
x	-1.466165414	0.094310214	-15.5461996	$4.4832546 \cdot 10^{-6}$

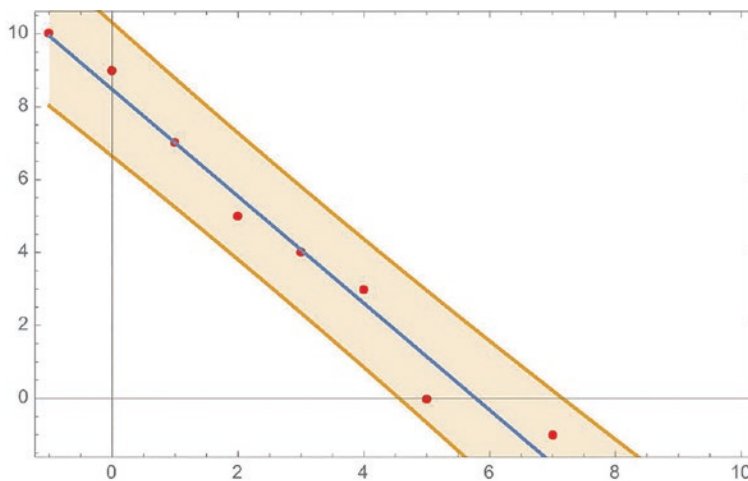
The coefficients are shown in the table. The first coefficient is the ordinate to the origin, and the coefficient associated with the e variable is the slope. The two coefficients have their respective standard errors. To know the confidence interval for the parameters, you write the `ParameterConfidenceIntervalTable` property.

```
In[90]:= model["ParameterConfidenceIntervalTable"]
Out[90]=
```

	Estimate	Standard Error	Confidence Interval
1	8.473684211	0.34167121	{7.63764488,9.30972355}
x	-1.466165414	0.094310214	{-1.69693419,-1.23539663}

The default confidence interval is 95%. With these confidence values, you can plot the points inside or outside this range (see Figure 6-25), extracting the values from the predictions and setting the option for the confidence interval to 0.95.

```
In[91]:= model[x];
model["SinglePredictionBands", ConfidenceLevel -> 0.95]; Show[
ListPlot[data, PlotStyle -> Red], Plot[{Model[x],
Model["SinglePredictionBands", ConfidenceLevel -> 0.95]}, {x, -1, 10},
Filling -> {2 -> {1}}], PlotRange -> {Automatic, {-1, 10}}, Frame -> True,
ImageSize -> 400]
Out[92]=
```



**Figure 6-25.** The filled region denotes the 95% confidence interval

Finally, to obtain the properties related to the sum of the squared errors, you use the `ANOVATable` property.

```
In[93]:= model["ANOVATable"]
```

```
Out[93]=
```

	DF	SS	MS	F-Statistic	P-Value
X	1	107.213346	107.213346	241.68432	$4.48325 \cdot 10^{-6}$
Error	6	2.6616541	0.44360902		
Total	7	109.8750000			

## Summary

This chapter covered the concepts and techniques for conducting statistical analysis using the Wolfram Language and how to perform linear adjustments (least squares, linear fit) through equations and implement specialized statistical functions—demonstrating that the Wolfram Language is an effective statistical tool. In addition, you also view the reference functions available in Mathematica for numerical and approximate calculations of descriptive statistics, random distributions, numbers, and sampling methods.

## CHAPTER 7

# Data Exploration

This chapter looks at the basics of data management through the Wolfram Data Repository online platform and its use in Mathematica. You also learn how data is viewed inside datasets and how to apply user functions and query commands.

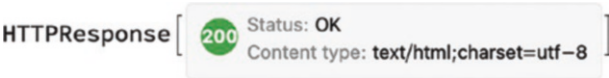
## Wolfram Data Repository

The Wolfram Data Repository is a data repository in the cloud. This data repository contains information from different categories, such as computer science, meteorology, agriculture, sports, text and literature education, and many more. Although this repository belongs to Wolfram Research, it is characterized by being in the public domain.

The Wolfram Data Repository consists of computable data selected, structured, and cured for direct use to perform numerical calculations, estimates, analysis, statistics, or demonstrations. The content hosted in this repository is data from many sources, globally known datasets, and publication data. All this information is designed so that any individual can access it globally. The Wolfram Data Repository system provides a data source that, in turn, also enables the storage of new information. The information stored in the repository is designed to directly implement the Wolfram Language.

As you saw in the data import section, you know whether the website is active by receiving an HTTP-type response, as shown in Figure 7-1.

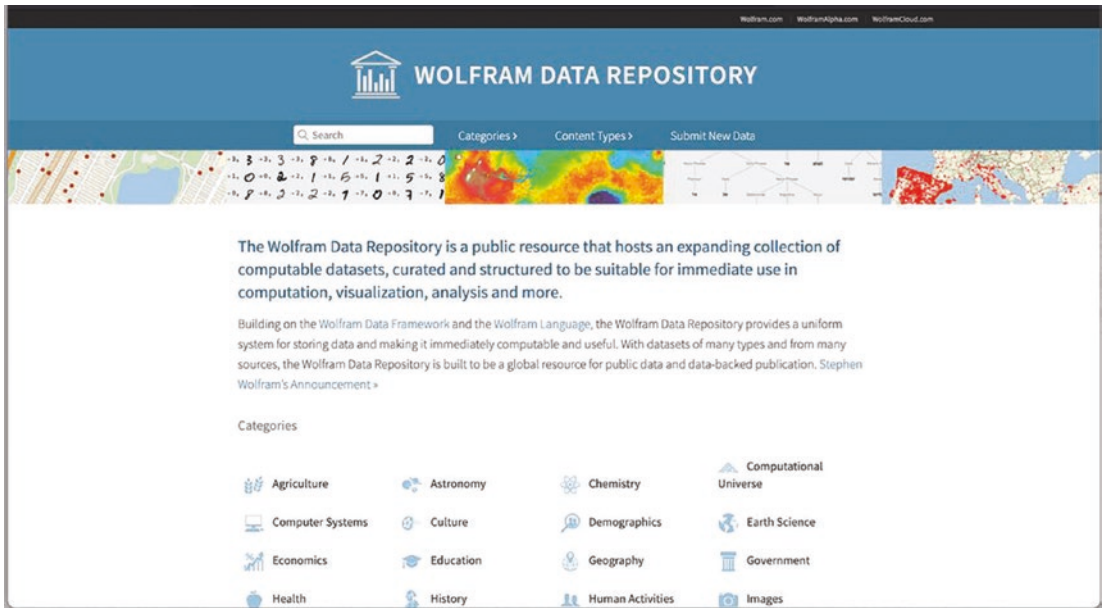
```
In[1]:= URLRead["https://datarepository.wolframcloud.com/"]  
Out[1]=
```



**Figure 7-1.** HTTP response object of the Wolfram Data Repository. As you can see, you have received a successful response.

## Wolfram Data Repository Website

To access this website, enter the following URL address in your favorite browser: <https://datarepository.wolframcloud.com>. Figure 7-2 shows the welcome page of the Wolfram Data Repository.

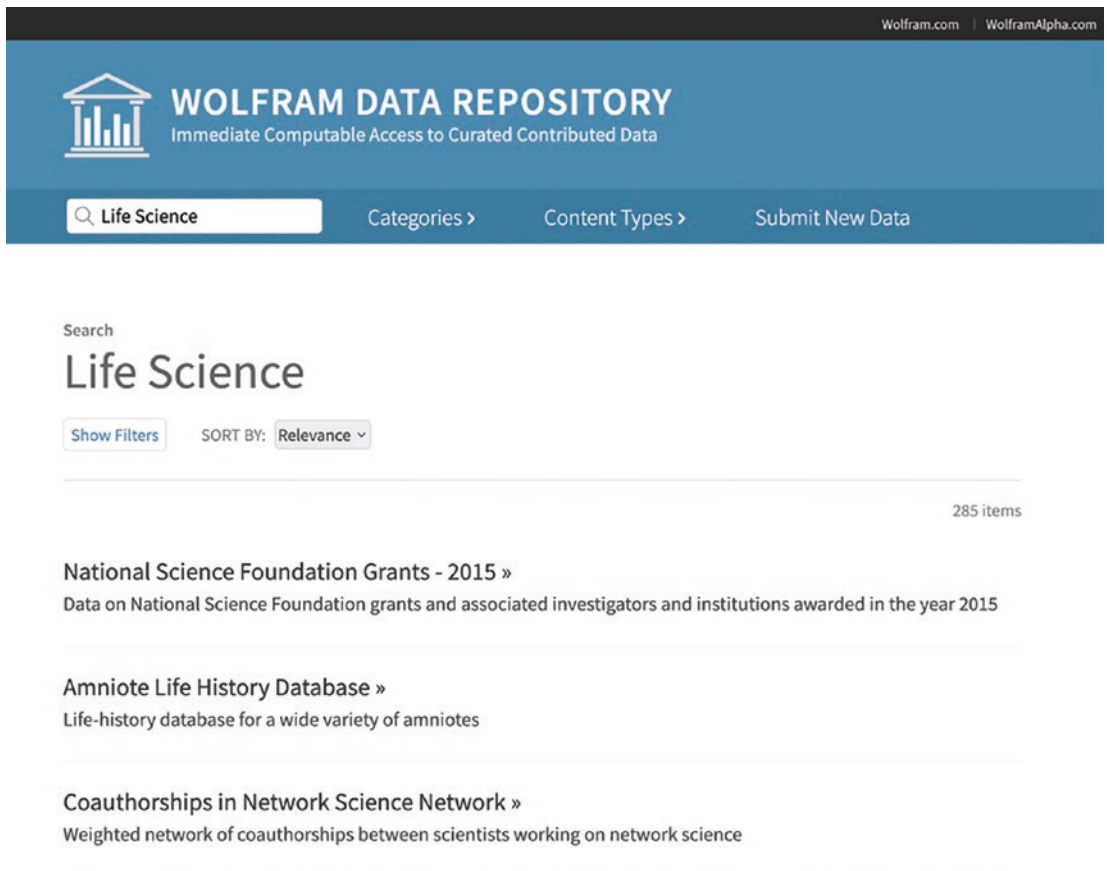


**Figure 7-2.** Wolfram Data Repository website

**Note** The images that appear are links that redirect to the dataset associated with that image.

Once the site is loaded, you see a menu of options to navigate the site, either by categories or data type. Within that menu, you find the different categories and data types: text, numerical data, images, and so forth. You also find the contact option,

custom searches, and Submit New Data among the menu options. The latter is the option that redirects to another page that displays the instructions for publishing and uploading new data to this repository. Scrolling down, you also see the existing categories and the data types. If so, there is the possibility to browse all resources by clicking the Browse All Resources link (bottom of the web page). To browse categories, you can choose the category from the menu or by clicking the category name at the initial site. Figure 7-3 shows what the site looks like once you have selected a category—in this case, Life Science.



**Figure 7-3.** *Life Science category of the Wolfram Data Repository*

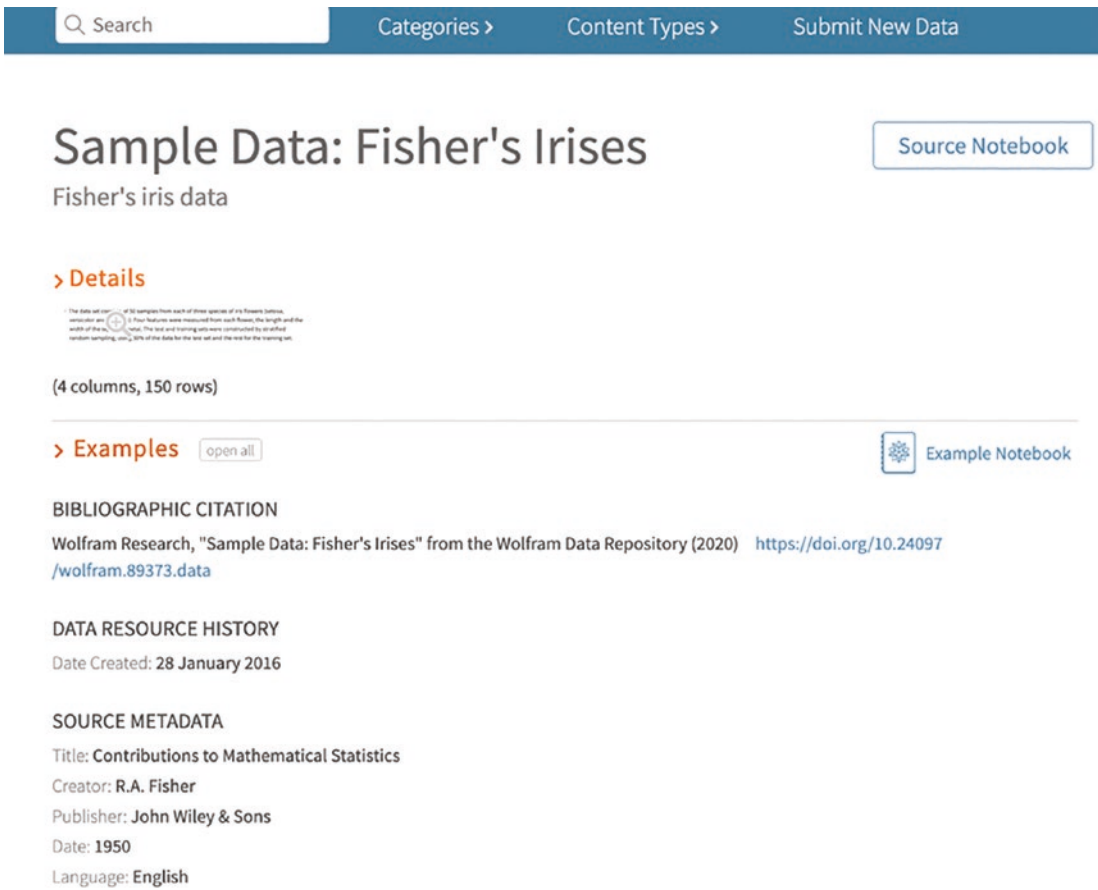
---

**Note** The same process is for navigating by data type. As new data is added, content is updated regularly.

---

# Selecting a Category

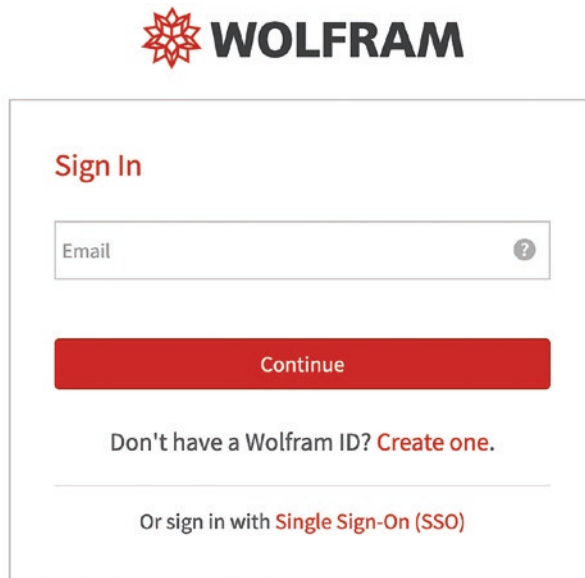
Each category shows the title, the number of elements in that category, and the option to filter the category’s contents by the data type. Regarding the content, each sample data type is displayed with its title, a small description of the data it contains, and the different tags associated with that sample data. For example, the image shows Fisher’s Irises’ known dataset. Once you select a sample dataset, it takes you to the site where the relevant information about that dataset is contained, as shown in Figure 7-4, where the Fisher’s Irises dataset is selected.



**Figure 7-4.** Fisher’s Irises dataset

When a sample dataset is selected, a brief description of the dataset is shown, as well as the different calculations that can be made and different formats to download the data or the notebook. Besides this, it also includes relevant information such as the

bibliographic citation, data resource history, and data source. In some instances, the data can be downloaded for different types of formats, such as comma-separated value (CSV), tab-separated value (TSV), JavaScript object notation (JSON), and others. Before starting to download data from the Wolfram Data Repository, it is necessary to have a Wolfram ID. This ID is an account that gives you access to the content of the Wolfram Data Repository in addition to other benefits, such as Wolfram One and Wolfram Alpha. To log in from Mathematica, head to the menu in Help ► Sign in, and a window appear like the one in Figure 7-5.



**Figure 7-5.** *Wolfram Cloud sign-in prompt*

In the new window, you enter your email and password to access the contents of the Wolfram Data Repository from Mathematica.

## Extracting Data from the Wolfram Data Repository

Let's start by looking at the information and properties of the Fisher's dataset; for this, you must retrieve the information through a ResourceObject. With ResourceObject (see Figure 7-6), you can now view the different properties of the published data by clicking the plus icon. Detailed information about the data is displayed, such as sample name, type, version, size of the data, and many more.

```
In[2]:= ResourceObject["Sample Data: Fisher's Irises"]
Out[2]=
```



**Figure 7-6.** *ResourceObject Fisher's Irises*

If you want to look at the properties of the resource object, enter the following code. This code gives you a list of properties that can be accessed and related to the data sample.

```
In[3]:= ResourceObject["Sample Data: Fisher's Irises"]["Properties"]
Out[3]= {AllVersions, AutoUpdate, Categories, ContentElementLocations,
ContentElements, ContentSize, ContentTypes, ContributorInformation,
DatedElementVersions, DefaultContentElement, Description, Details,
Documentation, DocumentationLink, DOI, DownloadedVersion, ExampleNotebook,
ExampleNotebookObject, Format, InformationElements, Keywords,
LatestUpdate, Name, Originator, Properties, PublisherUUID, ReleaseDate,
RepositoryLocation, ResourceLocations, ResourceType, SeeAlso, ShortName,
SourceMetadata, UUID, Version, VersionInformation, VersionsAvailable,
WolframLanguageVersionRequired}
```

Knowing the list of properties related to information, you can now download from Mathematica the exercise notebook of the data sample.

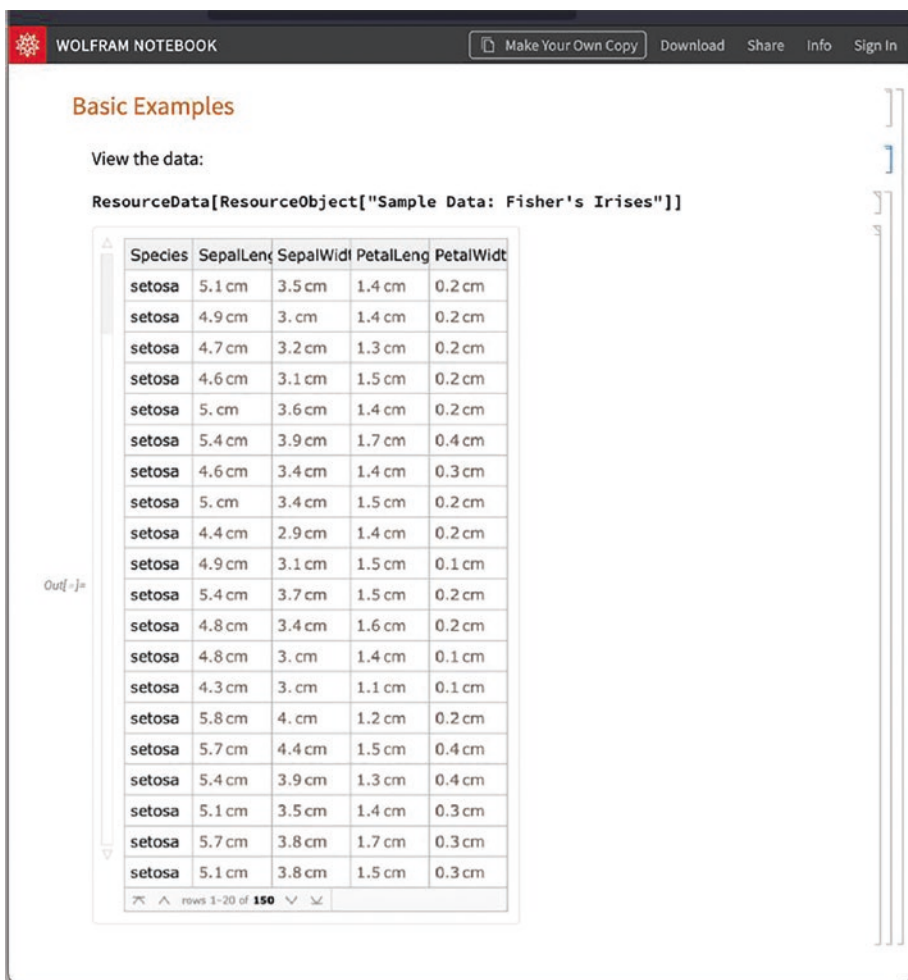
```
In[4]:=ResourceObject["Sample Data: Fisher's Irises"]["ExampleNotebook"]
Out[4]= NotebookObject[Sample Data: Fisher's Irises | Example Notebook]
```



Once you finish evaluating the code, it automatically opens the new notebook. If you want to operate the notebook from the cloud, you can type `NotebookObject`. This output gives you back a cloud-like object associated with a hyperlink.

```
In[5]:= ResourceObject["Sample Data: Fisher's Irises"]
["ExampleNotebookObject"]
Out[5]= CloudObject[https://www.wolframcloud.com/obj/5e59b79e-d95e-4f6f-
a7c8-f1276ba17be2]
```

If you press the link of the new notebook, it opens the Internet browser and shows you that it is in the Wolfram Cloud. Figure 7-7 shows this.



WOLFRAM NOTEBOOK

Make Your Own Copy Download Share Info Sign In

### Basic Examples

View the data:

`ResourceData[ResourceObject["Sample Data: Fisher's Irises"]]`

Species	SepalLength	SepalWidth	PetalLength	PetalWidth
setosa	5.1 cm	3.5 cm	1.4 cm	0.2 cm
setosa	4.9 cm	3. cm	1.4 cm	0.2 cm
setosa	4.7 cm	3.2 cm	1.3 cm	0.2 cm
setosa	4.6 cm	3.1 cm	1.5 cm	0.2 cm
setosa	5. cm	3.6 cm	1.4 cm	0.2 cm
setosa	5.4 cm	3.9 cm	1.7 cm	0.4 cm
setosa	4.6 cm	3.4 cm	1.4 cm	0.3 cm
setosa	5. cm	3.4 cm	1.5 cm	0.2 cm
setosa	4.4 cm	2.9 cm	1.4 cm	0.2 cm
setosa	4.9 cm	3.1 cm	1.5 cm	0.1 cm
setosa	5.4 cm	3.7 cm	1.5 cm	0.2 cm
setosa	4.8 cm	3.4 cm	1.6 cm	0.2 cm
setosa	4.8 cm	3. cm	1.4 cm	0.1 cm
setosa	4.3 cm	3. cm	1.1 cm	0.1 cm
setosa	5.8 cm	4. cm	1.2 cm	0.2 cm
setosa	5.7 cm	4.4 cm	1.5 cm	0.4 cm
setosa	5.4 cm	3.9 cm	1.3 cm	0.4 cm
setosa	5.1 cm	3.5 cm	1.4 cm	0.3 cm
setosa	5.7 cm	3.8 cm	1.7 cm	0.3 cm
setosa	5.1 cm	3.8 cm	1.5 cm	0.3 cm

Out[5]=

rows 1-20 of 150

**Figure 7-7.** Fisher's Irises data sample, open from the Wolfram Cloud

To access the original sample data site from Mathematica, enter **Documentation**, which gives you a URL object that you can enter by clicking the double chevron icon.

```
In[6]:= ResourceObject["Sample Data: Fisher's Irises"] ["Documentation"]  
Out[6]=URL[https://datarepository.wolframcloud.com/resources/  
b7f632f7-9c5f-4ad4-a73e-446cc2656f64/]
```

## Accessing Data Inside Mathematica

The same initiative is applied to downloading the data using the `ResourceData` to the object resource. With `ResourceData`, you access the contents of the specified resource; in this case, it is the Fisher's Irises data sample (see Figure 7-8).

```
In[7]:= ResourceData[ResourceObject["Sample Data: Fisher's Irises"]]  
Out[7]=
```



Species	SepalLength	SepalWidth	PetalLength	PetalWidth
setosa	5.1 cm	3.5 cm	1.4 cm	0.2 cm
setosa	4.9 cm	3. cm	1.4 cm	0.2 cm
setosa	4.7 cm	3.2 cm	1.3 cm	0.2 cm
setosa	4.6 cm	3.1 cm	1.5 cm	0.2 cm
setosa	5. cm	3.6 cm	1.4 cm	0.2 cm
setosa	5.4 cm	3.9 cm	1.7 cm	0.4 cm
setosa	4.6 cm	3.4 cm	1.4 cm	0.3 cm
setosa	5. cm	3.4 cm	1.5 cm	0.2 cm
setosa	4.4 cm	2.9 cm	1.4 cm	0.2 cm
setosa	4.9 cm	3.1 cm	1.5 cm	0.1 cm
setosa	5.4 cm	3.7 cm	1.5 cm	0.2 cm
setosa	4.8 cm	3.4 cm	1.6 cm	0.2 cm
setosa	4.8 cm	3. cm	1.4 cm	0.1 cm
setosa	4.3 cm	3. cm	1.1 cm	0.1 cm
setosa	5.8 cm	4. cm	1.2 cm	0.2 cm
setosa	5.7 cm	4.4 cm	1.5 cm	0.4 cm
setosa	5.4 cm	3.9 cm	1.3 cm	0.4 cm
setosa	5.1 cm	3.5 cm	1.4 cm	0.3 cm
setosa	5.7 cm	3.8 cm	1.7 cm	0.3 cm
setosa	5.1 cm	3.8 cm	1.5 cm	0.3 cm

rows 1-20 of 150

**Figure 7-8.** *Fisher's Irises dataset object*

As shown in Figure 7-8, the returned object is a `ResourceData` to use with a `head` of the dataset. Performing a visual inspection of the data sample, you observe that it is a dataset of 150 values containing five columns: `Species`, `SepalLength`, `SepalWidth`, `PetalLength`, and `PetalWidth`. If you pay attention, you can see how the values of the `SepalLength`, `SepalWidth`, `PetalLength`, and `PetalWidth` columns are quantities. Moving further down the entire dataset, the species are divided into three categories: `setosa`, `versicolor`, and `virginica`. If you want to access the information related to the dataset,

you must do it through the resource object and retrieve it through a `ResourceData` form, as shown.

```
In[8]:=ResourceObject["Sample Data: Fisher's Irises"]
["ContentElements"]
Out[8]= {ColumnDescriptions, ColumnTypes, Content, DataType, Dimensions,
ObservationCount, RawData, Source, TrainingData,
TestData}
```

With the `ContentElements` property, you are accessing the elements of the data sample, which are the ones that appear within the resource object. `ContentElements` shows you the information associated with the sample data, such as column information, data source, training data, and test data—not to be confused with the properties of the resource object created, as it is not the same since you can construct a resource object for another associated name. To retrieve the information from the `ContentElements`, you must do it with `ResourceData`. This command gives you access to the contents of the data sample—in this case, the Fisher's Irises. Now, let's get the data type of the columns.

```
In[9]:= ResourceData[ResourceObject["Sample Data: Fisher's
Irises"],"ColumnTypes"]
Out[9]= {Numeric,Numeric,Numeric,Numeric,Categorical}
```

The second argument of the `ResourceData` command is the element you are looking for. Running the code mentioned above shows you that there are four data types: three numeric and one categorical. Using a pure function, you can obtain information in a single expression. If you add the `Column` command, it is possible to have a better view of the information.

```
In[10]:= Column[ResourceData[ResourceObject["Sample Data: Fisher's
Irises"],#]&/@{"ColumnDescriptions","Dimensions","Source"}]
Out[10]= {Sepal length in cm.,Sepal width in cm.,Petal length in cm.,Petal
width in cm.,Species of iris}
{150,4}
Fisher,R.A. "The use of multiple measurements in taxonomic problems"
Annual Eugenics, 7, Part II, 179-188 (1936); also in "Contributions to
Mathematical Statistics" (John Wiley, NY, 1950).
```

This way, you get to know the type of information in the columns, such as dimensions, which are 150 rows per four columns, and the data source.

## Data Observation and Querying

This section explains how to observe data inside a dataset. You use the Iris dataset, which has been extracted from the Wolfram Data Repository. Let's start by naming the data sample Fisher; this variable contains the dataset with quantities included.

```
In[11]:= fisher=ResourceData[ResourceObject["Sample Data: Fisher's  
Irises"]];
```

In the dataset, the numbers have units and magnitude. Having a dataset, you can perform endless processes, such as grouping the content by the category variable, which is the type of species. (This example accessed the dataset contained in the Fisher's variable.) Let's look at the data that includes each column grouped by species (see Figure 7-9).

```
In[12]:= fisher[GroupBy["Species"]]  
Out[12]=
```

	Species	SepalLength	SepalWidth	PetalLength	PetalWidth
setosa	setosa	5.1 cm	3.5 cm	1.4 cm	0.2 cm
	setosa	4.9 cm	3. cm	1.4 cm	0.2 cm
	setosa	4.7 cm	3.2 cm	1.3 cm	0.2 cm
	setosa	4.6 cm	3.1 cm	1.5 cm	0.2 cm
	setosa	5. cm	3.6 cm	1.4 cm	0.2 cm
	50 total >				
versicolor	versicolor	7. cm	3.2 cm	4.7 cm	1.4 cm
	versicolor	6.4 cm	3.2 cm	4.5 cm	1.5 cm
	versicolor	6.9 cm	3.1 cm	4.9 cm	1.5 cm
	versicolor	5.5 cm	2.3 cm	4. cm	1.3 cm
	versicolor	6.5 cm	2.8 cm	4.6 cm	1.5 cm
	50 total >				
virginica	virginica	6.3 cm	3.3 cm	6. cm	2.5 cm
	virginica	5.8 cm	2.7 cm	5.1 cm	1.9 cm
	virginica	7.1 cm	3. cm	5.9 cm	2.1 cm
	virginica	6.3 cm	2.9 cm	5.6 cm	1.8 cm
	virginica	6.5 cm	3. cm	5.8 cm	2.2 cm
	50 total >				

**Figure 7-9.** *Iris data grouped by species*

Let’s look at how the data is divided into three categories: setosa, versicolor, and virginica. Each category contains a number 50 at the end of the Species column of each category. This means that there are 50 more rows in addition to those shown, making a total of 50 for each category, which is 150 rows in total, which matches the number of 150 you review the dimensions of the sample data.

In the meantime, clicking one of the categories shows you the columns for that category alone, as shown in Figure 7-10. The same happens if you select a specific column within a category—it shows only that column for that category; try it to see what happens. There is also the possibility to click any column, and this shows you only the chosen column for the three categories. This means that if you choose SepalLength, for example, you see the contents of that column for the three species, as shown in Figure 7-10.

☰ All All SepalLength

setosa	5.1 cm	4.9 cm	4.7 cm	4.6 cm	5. cm
	5.4 cm	4.6 cm	5. cm	4.4 cm	4.9 cm
	5.4 cm	4.8 cm	4.8 cm	4.3 cm	5.8 cm
	5.7 cm	5.4 cm	5.1 cm	5.7 cm	5.1 cm
	5.4 cm	5.1 cm	4.6 cm	5.1 cm	4.8 cm
	50 total >				
versicolor	7. cm	6.4 cm	6.9 cm	5.5 cm	6.5 cm
	5.7 cm	6.3 cm	4.9 cm	6.6 cm	5.2 cm
	5. cm	5.9 cm	6. cm	6.1 cm	5.6 cm
	6.7 cm	5.6 cm	5.8 cm	6.2 cm	5.6 cm
	5.9 cm	6.1 cm	6.3 cm	6.1 cm	6.4 cm
	50 total >				
virginica	6.3 cm	5.8 cm	7.1 cm	6.3 cm	6.5 cm
	7.6 cm	4.9 cm	7.3 cm	6.7 cm	7.2 cm
	6.5 cm	6.4 cm	6.8 cm	5.7 cm	5.8 cm
	6.4 cm	6.5 cm	7.7 cm	7.7 cm	6. cm
	6.9 cm	5.6 cm	7.7 cm	6.3 cm	6.7 cm
	50 total >				

**Figure 7-10.** *SepalLength* column selected

It is possible to group by species and choose only the columns that contain numeric values. This helps if, for example, you want to visually inspect the dataset (see Figure 7-11).

```
In[13]:= Query[GroupBy[ Key["Species"] -> KeyTake[{"SepalLength",
"SepalWidth", "PetalLength", "PetalWidth"}]]][fisher]
Out[13]=
```

	SepalLength	SepalWidth	PetalLength	PetalWidth
setosa	5.1 cm	3.5 cm	1.4 cm	0.2 cm
	4.9 cm	3. cm	1.4 cm	0.2 cm
	4.7 cm	3.2 cm	1.3 cm	0.2 cm
	4.6 cm	3.1 cm	1.5 cm	0.2 cm
	5. cm	3.6 cm	1.4 cm	0.2 cm
	50 total ›			
versicolor	7. cm	3.2 cm	4.7 cm	1.4 cm
	6.4 cm	3.2 cm	4.5 cm	1.5 cm
	6.9 cm	3.1 cm	4.9 cm	1.5 cm
	5.5 cm	2.3 cm	4. cm	1.3 cm
	6.5 cm	2.8 cm	4.6 cm	1.5 cm
	50 total ›			
virginica	6.3 cm	3.3 cm	6. cm	2.5 cm
	5.8 cm	2.7 cm	5.1 cm	1.9 cm
	7.1 cm	3. cm	5.9 cm	2.1 cm
	6.3 cm	2.9 cm	5.6 cm	1.8 cm
	6.5 cm	3. cm	5.8 cm	2.2 cm
	50 total ›			

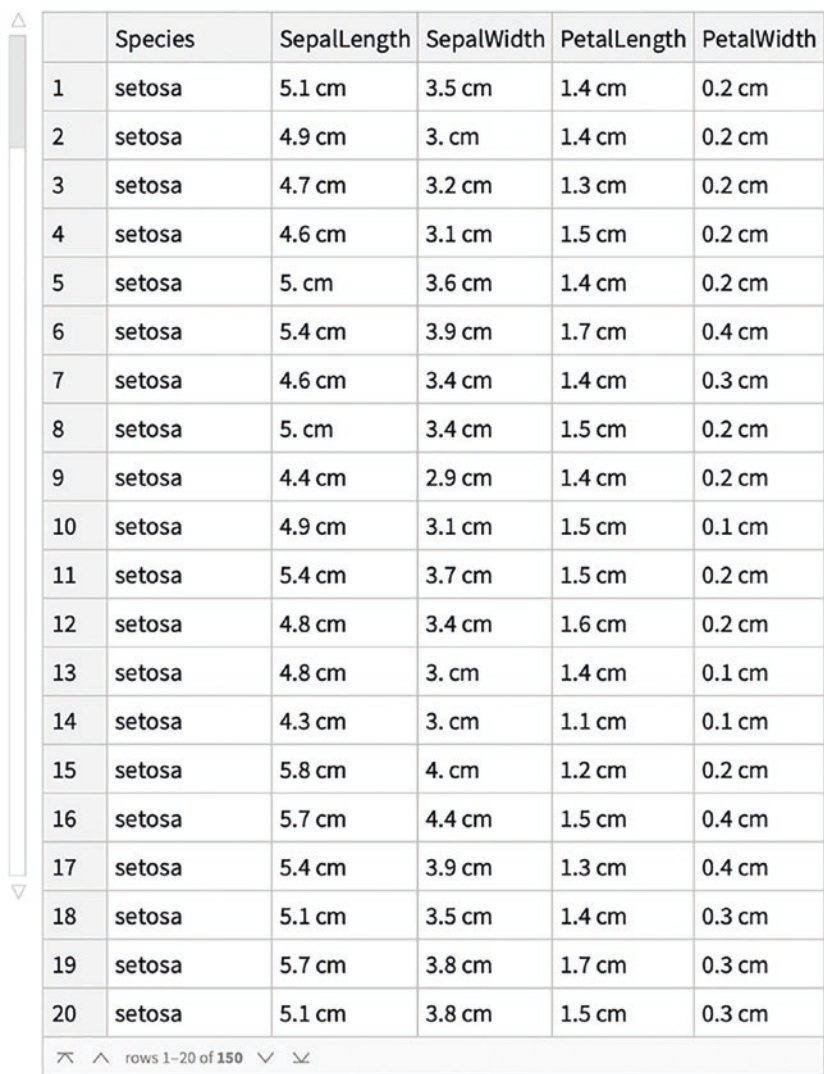
**Figure 7-11.** Dataset with the species column suppressed

In the latter code, you use the Key command to access the keys of the species column. Once these keys are accessed, you write a transformation rule so that each extracted key is assigned the associations extracted (KeyTake) from columns (SepalLength, SepalWidth, PetalLength, PetalWidth), then grouped and applied to Fisher’s dataset.

If you wanted to count the data elements in Fisher’s dataset, you could add an ID column as a label (see Figure 7-12) to list the data it contains. To achieve this, first, create an association with keys and values that go from 1 to the length of the dataset. Then, this instruction is applied to the dataset object Fisher’s, which adds the IDs as labels for the rows.

```
In[14]:= Query[AssociationThread[Range[Length@#]→Range[Length@#]]]
[fisher]&[fisher]
Out[14]=
```





	Species	SepalLength	SepalWidth	PetalLength	PetalWidth
1	setosa	5.1 cm	3.5 cm	1.4 cm	0.2 cm
2	setosa	4.9 cm	3. cm	1.4 cm	0.2 cm
3	setosa	4.7 cm	3.2 cm	1.3 cm	0.2 cm
4	setosa	4.6 cm	3.1 cm	1.5 cm	0.2 cm
5	setosa	5. cm	3.6 cm	1.4 cm	0.2 cm
6	setosa	5.4 cm	3.9 cm	1.7 cm	0.4 cm
7	setosa	4.6 cm	3.4 cm	1.4 cm	0.3 cm
8	setosa	5. cm	3.4 cm	1.5 cm	0.2 cm
9	setosa	4.4 cm	2.9 cm	1.4 cm	0.2 cm
10	setosa	4.9 cm	3.1 cm	1.5 cm	0.1 cm
11	setosa	5.4 cm	3.7 cm	1.5 cm	0.2 cm
12	setosa	4.8 cm	3.4 cm	1.6 cm	0.2 cm
13	setosa	4.8 cm	3. cm	1.4 cm	0.1 cm
14	setosa	4.3 cm	3. cm	1.1 cm	0.1 cm
15	setosa	5.8 cm	4. cm	1.2 cm	0.2 cm
16	setosa	5.7 cm	4.4 cm	1.5 cm	0.4 cm
17	setosa	5.4 cm	3.9 cm	1.3 cm	0.4 cm
18	setosa	5.1 cm	3.5 cm	1.4 cm	0.3 cm
19	setosa	5.7 cm	3.8 cm	1.7 cm	0.3 cm
20	setosa	5.1 cm	3.8 cm	1.5 cm	0.3 cm

rows 1-20 of 150

**Figure 7-12.** IDs added to the Fisher’s dataset

If you drag down the bar, you see that the counter reaches 150 elements.

You can use the Counts command if you don’t want to add an enumerated column to count the elements (see Figure 7-13).

```
In[15]:= Fisher[Counts,"Species"]
Out[15]=
```

setosa	50
versicolor	50
virginica	50

**Figure 7-13.** Counted elements on the dataset

This results in 50 data belonging to setosa, versicolor, and virginica. If you add them up, you get 150. You can also use the Query command, Query[Counts, "Species"] [Fisher].

Now, let's look at how to get the average of the three categories for each column. It would be possible if you knew the average of SepalLength, SepalWidth, PetalLength, and PetalWidth for the species, setosa, versicolor, and virginica, as exhibited in Figure 7-14.

```
In[16]:=Query[GroupBy[Key["Species"]→KeyTake[{"SepalLength","SepalWidth",
"PetalLength","PetalWidth"}]],Mean][fisher]
Out[16]=
```

	SepalLength	SepalWidth	PetalLength	PetalWidth
setosa	5.006 cm	3.428 cm	1.462 cm	0.246 cm
versicolor	5.936 cm	2.77 cm	4.26 cm	1.326 cm
virginica	6.588 cm	2.974 cm	5.552 cm	2.026 cm

**Figure 7-14.** Mean for the four columns, divided by species

But, if you want to get the average of the columns for all categories, one way to get it would be by applying Mean as a query to the number of columns in the entire dataset (see Figure 7-15).

```
In[17]:= Query[Mean][fisher[[All,2;;5]]]
Out[17]=
```

SepalLength	5.84333 cm
SepalWidth	3.05733 cm
PetalLength	3.758 cm
PetalWidth	1.19933 cm

**Figure 7-15.** Average values for the four columns of all species

---

**Note** The Mean command works with the quantities and returns the average to use as a quantity.

---

## Descriptive Statistics

This section demonstrates how to perform descriptive statistics of the Irises data and computations inside the dataset format and how to create custom grid formats. Let's start by building the function that calculates the maximum, minimum, mean, median, first, and third quartile.

```
In[18]:
stats[data_] :=
{{#["Max: ", Max@data]}},
{{#["Min: ", Min@data]}},
{{#["Mean: ", Mean@data]}},
{{#["Median: ", Median@data]}},
{{#["1st quartile: ", Quantile[data, 0.25]}},
{{#["3rd quartile: ", Quantile[data, 0.75]}},
}&[Row]
```

Now, apply the created function to each of the columns. This function is to get overall statistics for SepalLength, SepalWidth, PetalLength, and PetalWidth (see Figure 7-16).

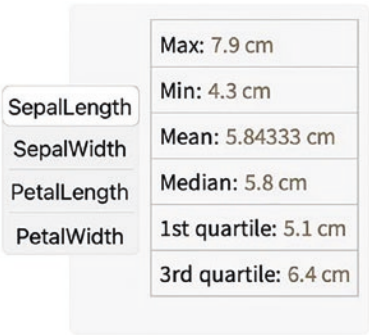
```
In[20]:= {{#1, #2, #3, #4}, {Fisher[Stats, #1], Fisher[Stats, #2], Fisher[Stats,
#3], Fisher[Stats, #4]}} &["SepalLength", "SepalWidth", "PetalLength",
"PetalWidth"] // Grid
Out[20]=
```

SepalLength	SepalWidth	PetalLength	PetalWidth
Max:7.9 cm	Max:4.4 cm	Max:6.9 cm	Max:2.5 cm
Min:4.3 cm	Min:2. cm	Min:1. cm	Min:0.1 cm
Mean:5.84333 cm	Mean:3.05733 cm	Mean:3.758 cm	Mean:1.19933 cm
Median:5.8 cm	Median:3. cm	Median:4.35 cm	Median:1.3 cm
1st quartile:5.1 cm	1st quartile:2.8 cm	1st quartile:1.6 cm	1st quartile:0.3 cm
3rd quartile:6.4 cm	3rd quartile:3.3 cm	3rd quartile:5.1 cm	3rd quartile:1.8 cm

**Figure 7-16.** Function Stats applied to each column

This also can be displayed in a compact form in a tab format with TabView (see Figure 7-17).

```
In[21]:= TabView[{#1->Fisher[Stats,#1],#2->Fisher[Stats,#2],#3->
Fisher[Stats,#3],#4->Fisher[Stats,#4]},ControlPlacement-> Left]&
["SepalLength","SepalWidth","PetalLength","PetalWidth"]
Out[21]=
```



**Figure 7-17.** Tabview format

With TabView, you create three tabs with the names of each column, which shows the values maximum, minimum, average, median, first, and third quartile; the columns are SepalLength, SepalWidth, PetalLength, and PetalWidth.

## Table and Grid Formats

An alternative is to create a table for each species. In this way, you better present the data and thus be able to read it properly. You extract the data by applying the Nest command. With this command, you can specify the number of times a command or function is applied; in this case, you apply it twice.

```
In[22]:= Short[Values[Nest[Normal, fisher, 2]]]
{sLall, sWall, pLall, pWall}=%[All, #]&/@{2, 3, 4, 5};
Out[22]//Short= {{setosa, 5.1cm, 3.5cm, 1.4cm, 0.2cm}, {setosa, 4.9cm, 3. cm, 1.4cm,
0.2cm}, <<146>>, {virginica, 6.2cm, 3.4cm, 5.4cm, 2.3cm}, {virginica, 5.9cm, 3. cm,
5.1cm, 1.8cm}}
```

Having the values of all species separated by columns, you create a list instead of a function, where the statistics are displayed according to each column, adding calculations such as variance, standard deviation, skewness, and kurtosis. Then, you assign the calculations in the DescriptiveStats variable.

```
In[23]:={Max[#], Min[#], Median[#], Mean[#], Variance[#], StandardDeviation[#], Skewness[#], Kurtosis[#], Quantile[#, 0.25], Quantile[#, .75]}&/@{sLall, sWall, pLall, pWall};
```

A table (see Figure 7-18) can be created with these calculations and adding the rows and column headings.

```
In[24]:= tableHeads={Style["Sepal Length", #1, ColorData["HTML"]
["Maroon"], #2, #3], Style["Sepal Width", #1, ColorData["HTML"]
["YellowGreen"], #2, #3], Style["Petal Length", #1, ColorData["HTML"]
["SteelBlue"], #2, #3], Style["Petal Width", #1, ColorData["HTML"]
["Orange"], #2, #3]}&["Title", Italic, 20];
tableRows={Style["Max", #1, #2], Style["Min", #1, #2], Style["Median", #1, #2],
Style["Mean", #1, #2], Style["Variance", #1, #2], Style["Standard\n Deviation", #1, #2],
Style["Skewness", #1, #2], Style["Kurtosis", #1, #2], Style["1st
quartile", #1, #2], Style["3rd quartile", #1, #2]}&["Text", Italic]; TableForm
[descriptiveStats, TableHeadings->{tableHeads, tableRows}]
Out[25]//TableForm=
```

	Max	Min	Median	Mean	Variance	Standard Deviation	Skewness	Kurtosis	1st quartile	3rd quartile
Sepal Length	7.9 cm	4.3 cm	5.8 cm	5.84333 cm	0.685694 cm <sup>2</sup>	0.828066 cm	0.311753	2.42643	5.1 cm	6.4 cm
Sepal Width	4.4 cm	2. cm	3. cm	3.05733 cm	0.189979 cm <sup>2</sup>	0.435866 cm	0.315767	3.18098	2.8 cm	3.3 cm
Petal Length	6.9 cm	1. cm	4.35 cm	3.758 cm	3.11628 cm <sup>2</sup>	1.7653 cm	-0.272128	1.60446	1.6 cm	5.1 cm
Petal Width	2.5 cm	0.1 cm	1.3 cm	1.19933 cm	0.581006 cm <sup>2</sup>	0.762238 cm	-0.101934	1.66393	0.3 cm	1.8 cm

Figure 7-18. Table showing descriptive statistics by the four features

Note that the statistics are calculated with their units, except for skewness and kurtosis, since, by definition, they are dimensionless. However, you can create a better structure from Grid because it is possible to add dividers like a spreadsheet format. To do this, you add the TableRows to the data and then apply a transpose so that each calculated statistic is with its respective name. Subsequently, you add the column titles.

```
In[26]:=
Transpose[Prepend[descriptiveStats,tableRows]];
{" ",Style["Sepal Length",#1, ColorData["HTML"]["Maroon"],#2,#3],
Style["Sepal Width",#1,ColorData["HTML"]["YellowGreen"],#
2,#3],Style["Petal Length",#1, ColorData["HTML"]["SteelBlue"],#2,
#3], Style["Petal Width",#1,ColorData["HTML"]["Orange"],
#2,#3]}&["Title",Italic,20];
newTable=Prepend[%%,%];
```

Next, create the table as a spreadsheet (see Figure 7-19).

```
In[27]:= Grid[newTable,ItemSize->{{None,Scaled[0.11], Scaled[0.11],
Scaled[0.11]}},Background->{{LightGray},None}, Dividers->{{False},
{1,2,3,4,5,6,7,8,9,10,11->True,-2->Blue}}, Alignment->Center]
Out[27]=
```

	<i>Sepal Length</i>	<i>Sepal Width</i>	<i>Petal Length</i>	<i>Petal Width</i>
<i>Max</i>	7.9 cm	4.4 cm	6.9 cm	2.5 cm
<i>Min</i>	4.3 cm	2. cm	1. cm	0.1 cm
<i>Median</i>	5.8 cm	3. cm	4.35 cm	1.3 cm
<i>Mean</i>	5.84333 cm	3.05733 cm	3.758 cm	1.19933 cm
<i>Variance</i>	0.685694 cm <sup>2</sup>	0.189979 cm <sup>2</sup>	3.11628 cm <sup>2</sup>	0.581006 cm <sup>2</sup>
<i>Standard Deviation</i>	0.828066 cm	0.435866 cm	1.7653 cm	0.762238 cm
<i>Skewness</i>	0.311753	0.315767	-0.272128	-0.101934
<i>Kurtosis</i>	2.42643	3.18098	1.60446	1.66393
<i>1st quartile</i>	5.1 cm	2.8 cm	1.6 cm	0.3 cm
<i>3rd quartile</i>	6.4 cm	3.3 cm	5.1 cm	1.8 cm

**Figure 7-19.** Grid view of the descriptive statistics

To build the table for each species, you must first separate the dataset by species with the Cases command. You should use Cases since it allows you to work with patterns. First, write the code to extract the raw data. Instead of using Short, use Shallow to suppress the 150 values.

```
In[28]:= Shallow[Values[Nest[Normal, fisher, 2]], 1]
Out[28]//Shallow= {<<150>>}
```

Create the table for the versicolor species, extract the values for versicolor, and store the values of the columns in the SLVersi, SWVersi, PLVersi, and PWVersi variables.

```
In[29]:= Shallow[Cases[%, {"versicolor", __}], 1]
{slVersi, swVersi, plVersi, pwVersi} = %[[All, #]] & /@ {2, 3, 4, 5};
Out[29]//Shallow= {<<50>>}
```

Next, repeat the process to calculate the statistics, but instead of the white space, add the name “Versicolor” in the Style text, to distinguish that the table belongs to the versicolor species.

```
In[30]:= tableRows;
{Max[#], Min[#], Median[#], Mean[#], Variance[#], StandardDeviation[#], Skewness
[#], Kurtosis[#], Quantile[#, 0.25], Quantile[#, .75]} & /@ {slVersi, swVersi,
plVersi, pwVersi};
descriptiveStats2 = Prepend[%, %];
```

```
Transpose[descriptiveStats2];
{Style["Versicolor", "Text", Red, Italic, 20], Style["Sepal Length", #1, ColorData["HTML"]["Maroon"], #2, #3], Style["Sepal Width", #1, ColorData["HTML"]["YellowGreen"], #2, #3], Style["Petal Length", #1, ColorData["HTML"]["SteelBlue"], #2, #3], Style["Petal Width", #1, ColorData["HTML"]["Orange"], #2, #3]} &["Title", Italic, 20];
newTable2=Prepend[%%,%];
```

Next, build the table (see Figure 7-20) for the species versicolor.

```
In[31]:= Grid[newTable2, ItemSize-> {{None, Scaled[0.11], Scaled[0.11], Scaled[0.11]}}, Background->{{LightGray}, None}, Dividers-> {{False}, {1,2,3,4,5,6,7,8,9,10,11->True, -2->Blue}}, Alignment-> Center]
Out[31]=
```

<i>Versicolor</i>	<i>Sepal Length</i>	<i>Sepal Width</i>	<i>Petal Length</i>	<i>Petal Width</i>
Max	7. cm	3.4 cm	5.1 cm	1.8 cm
Min	4.9 cm	2. cm	3. cm	1. cm
Median	5.9 cm	2.8 cm	4.35 cm	1.3 cm
Mean	5.936 cm	2.77 cm	4.26 cm	1.326 cm
Variance	0.266433 cm <sup>2</sup>	0.0984694 cm <sup>2</sup>	0.220816 cm <sup>2</sup>	0.0391061 cm <sup>2</sup>
Standard Deviation	0.516171 cm	0.313798 cm	0.469911 cm	0.197753 cm
Skewness	0.10219	-0.351867	-0.588159	-0.0302363
Kurtosis	2.40117	2.55173	2.9256	2.51217
1st quartile	5.6 cm	2.5 cm	4. cm	1.2 cm
3rd quartile	6.3 cm	3. cm	4.6 cm	1.5 cm

**Figure 7-20.** Descriptive stats for the versicolor species

You have only done this for the versicolor species; the same process is performed for each species. For example, if you choose Cases with the other species, you would change the text to the corresponding species.

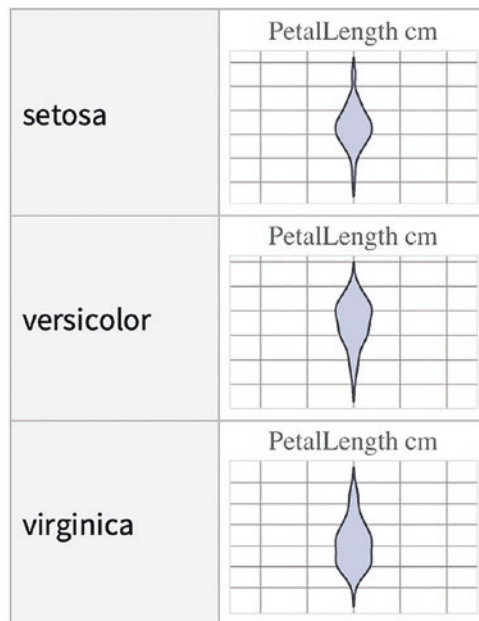


## Dataset Visualization

Having viewed the capabilities of the Wolfram Language to perform descriptive statistics within the dataset, statistical charts can be implemented inside the dataset format, as you see in this fragment.

You can have a better perspective from graphs; you use the dataset format (see Figure 7-21) to display the graphs by their species.

```
In[32]:= fisher[GroupBy["Species"],DistributionChart[#,Plot
tTheme-> "Classic",PlotLabel->"PetalLength cm",GridLines->
Automatic]&,"PetalLength"]
Out[32]=
```



**Figure 7-21.** *Distribution chart plot*

You can perform the same process but for the box whiskers plot (see Figure 7-22), but choose another column.

```
In[33]:= fisher[GroupBy["Species"],BoxWhiskerChart[#, "Outliers",PlotThe
me-> "Detailed",ChartLabels->Placed[{"SepalLength cm"},Above],BarOrigin->
Right,ChartStyle->Blue]&,"SepalLength"]
Out[33]=
```

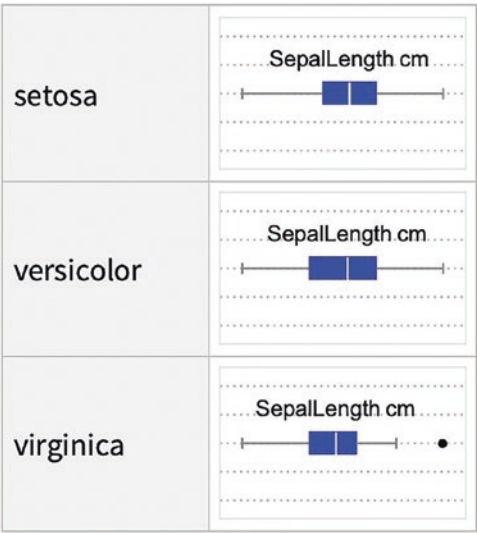


Figure 7-22. Box whiskers plot

If the specie is clicked, it amplfy the graph (see Figure 7-23).

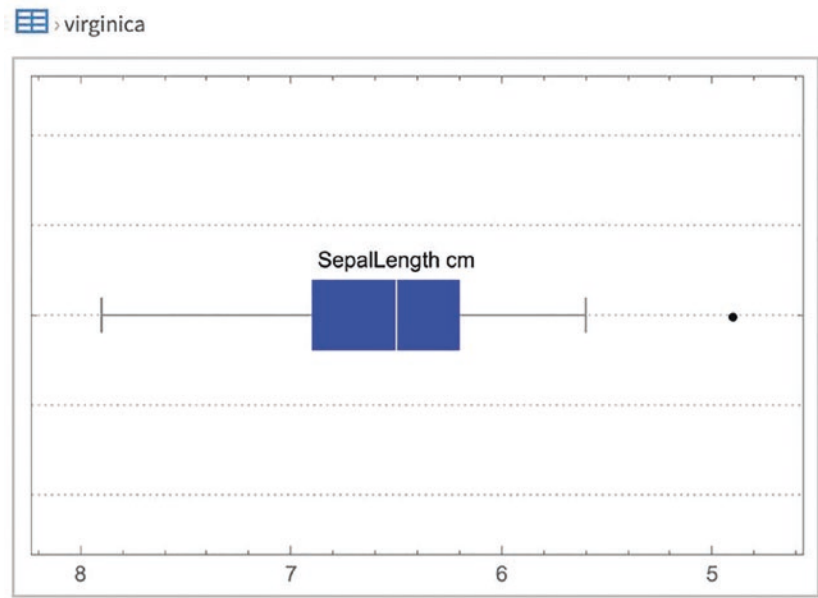
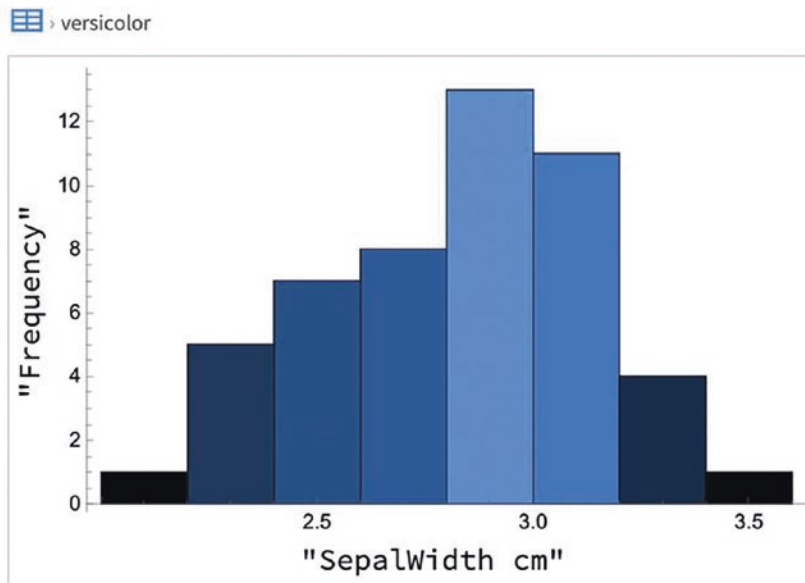


Figure 7-23. Box whiskers plot for virginica species

The same applies to histograms. When the graph is extensive, it appears suppressed within the dataset, but you can still select it, as shown in Figure 7-24.

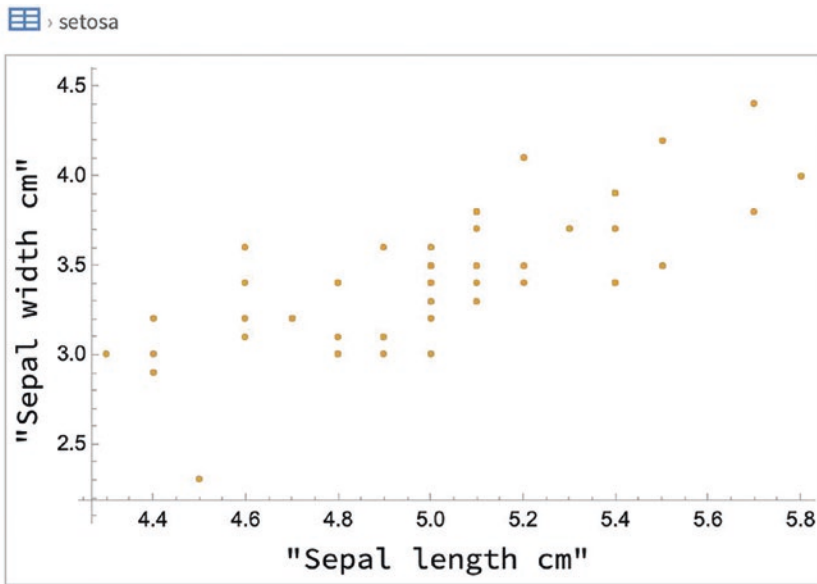
```
In[34]:=fisher[GroupBy["Species"], Labeled[Histogram[#, ColorFunction ->
(Hue[3/5, 2/3, #] &)], {Rotate["Frequency", 90 Degree], "SepalWidth cm"},
{Left, Bottom}] &, "SepalWidth"]
Out[34]=
```



**Figure 7-24.** Histogram plot for *versicolor*

Here, you show the 3D scatter plots for each species (see Figure 7-25) for sepal length (x) vs. sepal width (y).

```
In[35]:=Fisher[GroupBy["Species"], Labeled[ListPlot[{#, #}], {Rotate["Sepal
width cm", 90 Degree], "Sepal length cm"}, {Left, Bottom}] &,
{"SepalLength", "SepalWidth"}]
Out[35]=
```



**Figure 7-25.** 2D scatter plot

To return to the full dataset, click the dataset icon as with any other.

## Data Outside Dataset Format

The truth is that there is also the possibility of extracting the data crudely, as follows. You'll do this to have better data handling. You use the Short command since the list is quite long.

```
In[36]:= Short[ResourceData[ResourceObject["Sample Data: Fisher's
Iris"], "RawData"]]
Out[36]//Short= {<|Species->setosa, SepalLength-
>5.1cm, SepalWidth->3.5cm, PetalLength->1.4cm, PetalWidth-
>0.2cm|>, <|<<1>>|>, <<146>>, <|<<1>>|>, <|<<1>>|>}
```

With the data already extracted, you can get the values with the Values function and convert them to normal expressions.

```
In[37]:= Short[Normal[Values[%]]]
Out[37]//Short= {{setosa, 5.1cm, 3.5cm, 1.4cm, 0.2cm}, {setosa, 4.9cm, 3.cm, 1.4cm,
0.2cm}, <<146>>, {virginica, 6.2cm, 3.4cm, 5.4cm, 2.3cm}, {virginica, 5.9cm, 3.cm,
5.1cm, 1.8cm}}
```

With the help of `MapAt`, you can extract the magnitudes of the quantities. The `MapAt` command lets you choose where to apply the `Quantity` function. You decided to apply it to all rows with `All`, but only from columns 2 to 4, which is where the quantities are located.

```
In[38]:= Short[iris=MapAt[QuantityMagnitude,%,{All,2;;5}]]
Out[38]//Short={{setosa,5.1,3.5,1.4,0.2},<<148>>,{virginica,
5.9,3.,5.1,1.8}}
```

Why remove the units if calculations can be made with them? You extract the magnitudes for all quantities because they have the same order of magnitude (cm), so each calculation is in the same units, except if you make conversions or transformations to the data.

## 2D and 3D Plots

On the other hand, it is easier to manipulate lists with Wolfram Language. Having the data in the form of lists, you now plot the three columns in a box plot and a distribution graph (see Figure 7-26). You only choose the three columns.

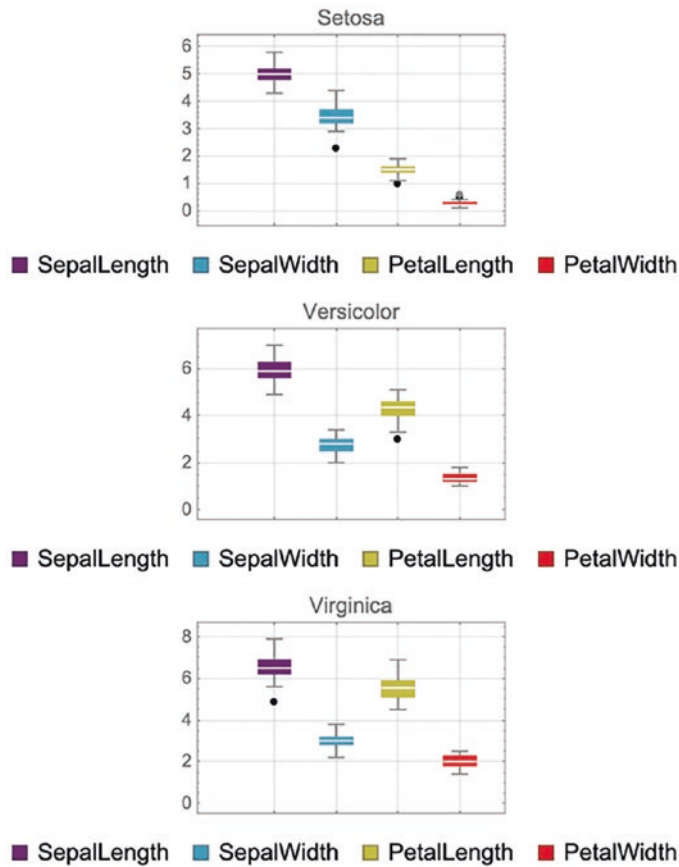
```
In[39]:=
Row[{BoxWhiskerChart[{iris[[All, #1]], iris[[All, #2]], iris[[All,
#3]], iris[[All, #4]]}, "Outliers", PlotRange -> Automatic, FrameTicks
-> True, ChartStyle -> "SandyTerrain", PlotLabel -> "All Species",
GridLines -> Automatic, ChartLegends -> Placed[{"SepalLength",
"SepalWidth", "PetalLength", "PetalWidth"}, Bottom], ImageSize -> Small],
DistributionChart[{iris[[All, #1]], iris[[All, #2]], iris[[All, #3]],
iris[[All, #4]]}, PlotRange -> Automatic, FrameTicks -> True, ChartStyle ->
"SouthwestColors", PlotLabel -> "All Species", ChartLegends ->
Placed[{"SepalLength", "SepalWidth", "PetalLength", "PetalWidth"}, Bottom],
PlotTheme -> "Detailed", GridLines -> Automatic, ImageSize -> Small]]] &[2,
3, 4, 5]
Out[39]=
```



**Figure 7-26.** Box whiskers plot and distribution chart for all species

To improve this, let's graph for each species. You use Cases to separate the list with their respective species (see Figure 7-27).

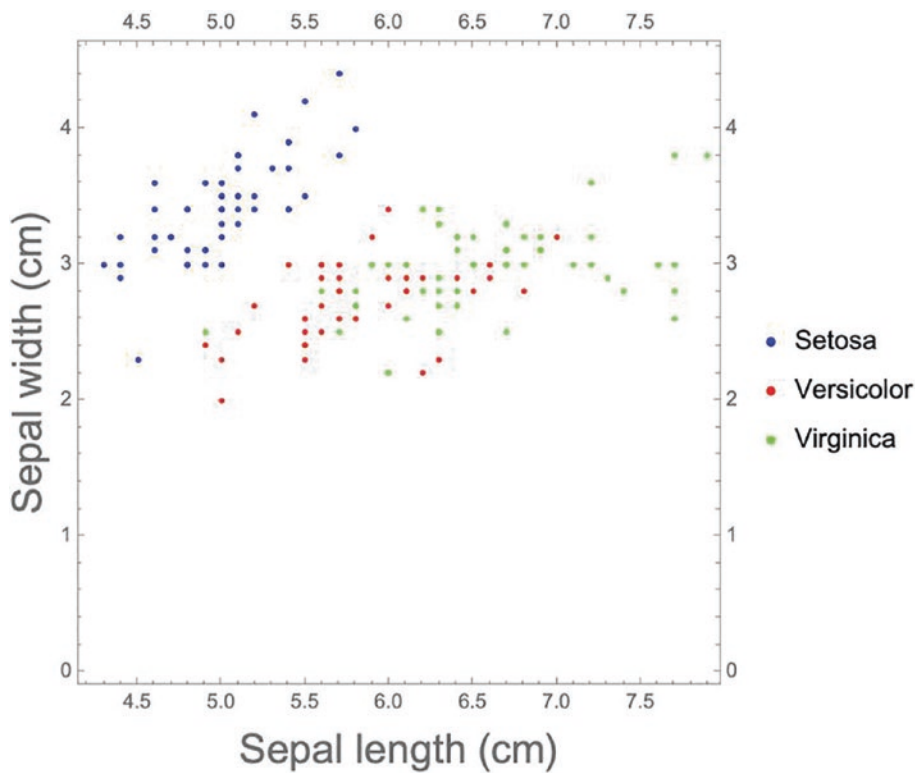
```
In[40]:= Short[setosa=Cases[iris,{"setosa",__}]];
Short[versi=Cases[iris,{"versicolor",__}]];
Short[virgin=Cases[iris,{"virginica",__}]];
Column@{BoxWhiskerChart[{setosa[[All,#1]],setosa[[All,#2]],setosa[[All,#3]],
setosa[[All,#4]]}, "Outliers", PlotRange->Automatic, FrameTicks->True,
ChartStyle->"Rainbow", PlotLabel->"Setosa", ChartLegends->Placed
[{"SepalLength", "SepalWidth", "PetalLength", "PetalWidth"}, Bottom],
GridLines->Automatic], BoxWhiskerChart[{versi[[All,#1]],versi[[All,#2]],
versi[[All,#3]],versi[[All,#4]]}, "Outliers", PlotRange->Automatic,
FrameTicks->True, ChartStyle->"Rainbow", PlotLabel->"Versicolor", ChartLegends->
Placed[{"SepalLength", "SepalWidth", "PetalLength", "PetalWidth"}, Bottom],
GridLines->Automatic], BoxWhiskerChart[{virgin[[All,#1]],virgin[[All,#2]],v
irgin[[All,#3]],virgin[[All,#4]]}, "Outliers", PlotRange->Automatic, FrameTicks->
True, ChartStyle->"Rainbow", PlotLabel->"Virginica", ChartLegends-> Placed
[{"SepalLength", "SepalWidth", "PetalLength", "PetalWidth"}, Bottom], GridLines->
Automatic]
}&[2,3,4,5]
Out[40]=
```



**Figure 7-27.** Box whiskers plot for every species with the four features

In addition, you can join the scatter plots of sepal width vs. sepal length for all species (see Figure 7-28).

```
In[41]:= ListPlot[{setosa[[All, {2, 3}]], versi[[All, {2, 3}]],
virgin[[All, {2, 3}]]}, FrameTicks -> All, Frame -> True,
AspectRatio -> 1, PlotStyle -> {Blue, Red, Green},
FrameLabel -> {Style["Sepal length (cm)", FontSize -> 20],
Style["Sepal width (cm)", FontSize -> 20]}, PlotLegends -> {"Setosa",
"Versicolor", "Virginica"}]
Out[41]=
```

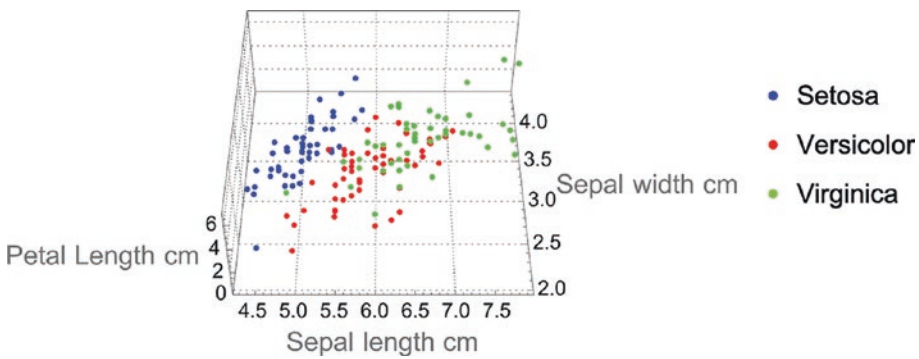


**Figure 7-28.** 2D scatter plot for all species of the first two features

Or you can make a 3D scatter plot with three features (see Figure 7-29).

```
In[42]:= ListPointPlot3D[{setosa[[All, {2, 3, 4}]], versi[[All, {2, 3,
4}]], virgin[[All, {2, 3, 4}]]}, Ticks -> All, AspectRatio -> 1,
PlotStyle -> {Blue, Red, Green}, AxesLabel -> {Style["Sepal length cm",
FontSize -> 13], Style["Sepal width cm", FontSize -> 13],
Style["Petal Length cm", FontSize -> 13]}, PlotLegends -> {"Setosa",
"Versicolor", "Virginica"}, PlotTheme -> "Detailed", ViewPoint ->
{0, -3, 3}]
Out[42]=
```





**Figure 7-29.** 3D scatter plot of three features for every species

Now, when you have finished working with the resource object, you need to delete it so that the local cache of the resource is removed correctly.

```
In[43]:=Clear[fisher]
DeleteObject[ResourceObject["Sample Data: Fisher's Irises"]]
```

## Summary

This chapter explored data exploration using the Wolfram Language. It starts by covering the Wolfram Data Repository, where instructions to navigate the website and select appropriate data categories effortlessly are addressed. The chapter continues to guide by showing how to extract data from the repository, offering insights on accessing, filtering, and observing the data within Mathematica. Additionally, the descriptive statistics section provides the reader with an understanding of table and grid formats. By the end of the chapter, it assists in mastering the visualization of datasets for 2D and 3D plots.

## CHAPTER 8

# Machine Learning with the Wolfram Language

This chapter introduces the gradient descent algorithm as an optimization method for linear regression; the corresponding computations are shown, as well as the concept of the learning curve of the model. Later, you see how to use the specialized functions of the Wolfram Language for machine learning, such as `Predict`, `Classify`, and `ClusterClassify`, in the case of linear regression, logistic regression, and cluster search. The different objects and results generated by these functions and the metrics to measure the model are shown for these functions. In each case, the parts of the model that are fundamental for the correct construction using the Wolfram Language are explained. This part of the book uses examples of known datasets such as the Fisher's Irises, Boston Homes, and Titanic datasets.

## Gradient Descent Algorithm

The gradient descent is an optimization algorithm that finds the minimum of a function through an iterative process. To build the process, the squared error loss function is minimized with the linear model hypothesis of the shape of  $(x_j) = \theta_0 + \theta_1 * x_j$ , around the point  $x_j$ . The following expression gives the loss function.

$$J(\theta) = \frac{1}{2 * N} \sum_{j=1}^N \left( f(x_j) - y_j \right)^2$$

$J(\theta)$  is the cost function,  $N$  is the number of observations,  $f(x_j)$  is the predicted output for observation  $j$ , and  $y_j$  is the actual output for observation  $j$ . The iterative process of the algorithm consists of calculating the coefficients until convergence is obtained. The following expressions give the coefficients.

$$\theta_0^{i+1} = \theta_0^i - \alpha \left( \frac{1}{N} \sum_{j=1}^N (\theta_0^i + \theta_1^i * x_j - y_j) \right)$$

$$\theta_1^{i+1} = \theta_1^i - \alpha \left( \frac{1}{N} \sum_{j=1}^N (\theta_0^i + \theta_1^i * x_j - y_j) * x_j \right)$$

Here,  $\theta_0^{i+1}$  and  $\theta_1^{i+1}$  represent the updated parameters after the  $i+1$  th iteration.  $\theta_0^i$  and  $\theta_1^i$  indicate their current values at the  $i$ th iteration,  $\alpha$  is the learning rate, a hyperparameter for updating  $\theta_0$  and  $\theta_1$ , that minimizes error during the learning process. At the same time,  $N$  is the total number of dataset observations.  $x_j$  and  $y_j$  are the  $j$ th observations of the independent and dependent variables in the dataset, respectively. The summations are obtained from partial derivatives concerning  $\theta_0$  and  $\theta_1$ . For more mathematical depth about the method and demonstrations, see *Artificial Intelligence: A Modern Approach* by Stuart Russell and Peter Norvig (Prentice Hall, 2010).

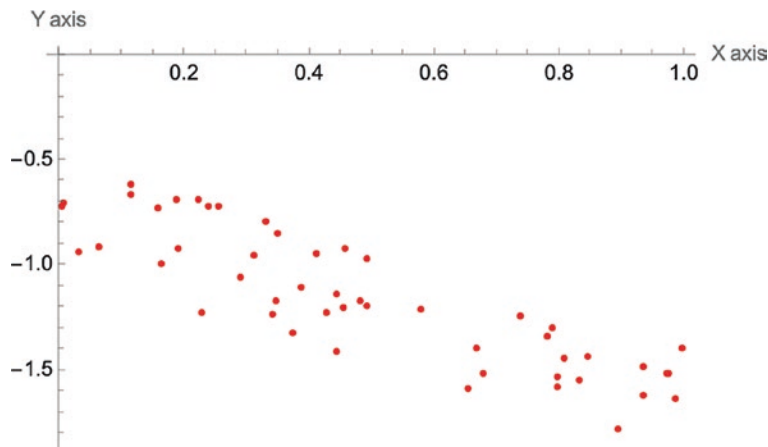
## Getting the Data

First, you define the data with the `RandomReal` function and establish a seed. This is to maintain the reproducibility of the data in case of practicing the same example.

```
In[1]:=
SeedRandom[888];
x=RandomReal[{0,1},50];
y=-1-x+0.6*RandomReal[{0,1},50];
```

Therefore, let's observe the data with a 2D scatter plot Figure 8-1.

```
In[4]:= ListPlot[Transpose[{x,y}],AxesLabel->{"X axis","Y
axis"},PlotStyle->Red]
Out[4]=
```



**Figure 8-1.** 2D scatter plot of the randomly generated data

## Algorithm Implementation

Let's now proceed to implement the algorithm with the Wolfram Language. The algorithm defines the constants, the number of iterations, and the learning rate. Then, you create two lists containing initial values of zero, in which the values of the coefficients for each iteration are stored. Later, you calculate the coefficients through a loop with `Table`, which does not end until you reach the number of iterations. In this case, you establish several iterations of 250 with a learning rate of 1.

```
In[5]:=
itt=250;(*Number of iterations*)
\[Alpha]=1;(*Learning rate*)
\[Theta]0=Range@@{0,itt};(* Array for values of Theta_0*)
\[Theta]1=Range@@{0,itt};(* Array for values of Theta_1*)
Table[{\[Theta]0[[i+1]]=\[Theta]0[[i]]-\[Alpha]/Length@x* Sum[(\[Theta]0[[i]]+\[Theta]1[[i]]* x[[j]]-y[[j]]),{j,1,Length@x}];
\[Theta]1[[i+1]]=\[Theta]1[[i]]-\[Alpha]/Length@x*Sum[(\[Theta]0[[i]]+\[Theta]1[[i]]*x[[j]]- y[[j]])* x[[j]],{j,1,Length@x}];},{i,1,itt}};
```

Since you have determined the calculation of the coefficients, you build the linear adjustment equation by constructing a function and using the coefficient values of the last iteration, which are in the previous position of the lists  $\theta_0$  y  $\theta_1$ .

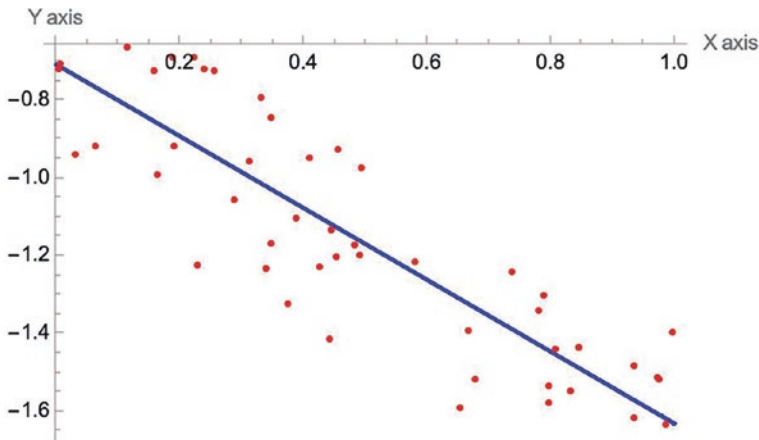
```
In[10]:= F[X_] := \[Theta]0[[Length@\[Theta]0]] + \[Theta]1[[Length@\[Theta]1]]*X
```

To know the shape of the best fit, you add the X variable as an argument. This gives you the form  $F(X) = \theta_0 + \theta_1 * X$ .

```
In[11]:= F[X]
Out[11]= -0.707789-0.923729 X
```

Look at how the line fits the data in Figure 8-2.

```
In[12]:= Show[{Plot[F[X],{X,0,1},PlotStyle->Blue,AxesLabel->{"X axis",
"Y axis"}],ListPlot[Transpose[{x,y}],PlotStyle->Red]]]
Out[12]=
```



**Figure 8-2.** Adjusted line to the data

Since you have built the linear model, you can make a graphical comparison of the variation of the learning rate with the number of iterations and the loss value given by the function  $J$ . But first, you must declare the loss function  $J$ . For the summation, you can either use the special symbols of sigma ( $\sum$ ) or write `Sum [expr, {i,imax}]`.

```
In[13]:= J[Theta0_, Theta1_] := 1/(2*Length[x])* Sum[ (Theta0 +
(Theta1*x[[i]]) - y[[i]])^2, {i, 1, Length@x}]
```

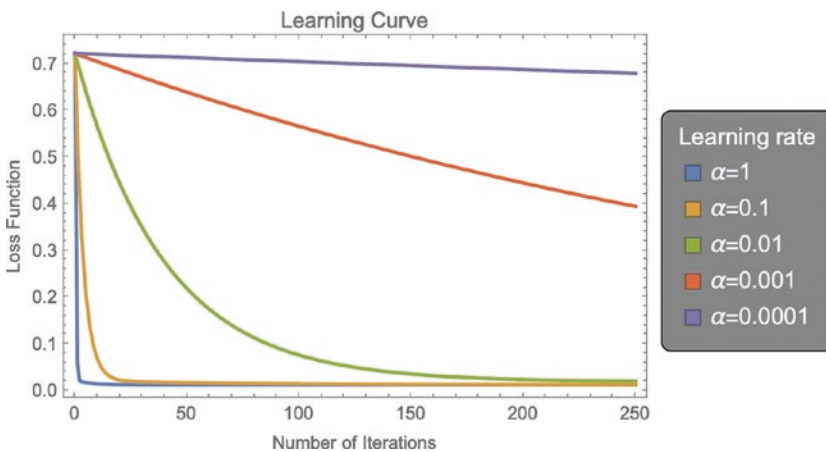
## Multiple Alphas

Having seen the previously constructed process, you can repeat the process for different alphas. Following is the graph of loss vs. each interaction for learning rate values of  $\alpha_1=1$ ,  $\alpha_2=0.1$ ,  $\alpha_3=0.01$ ,  $\alpha_4=0.001$ , and  $\alpha_5=0.0001$ , when repeating the process.

```
In[14]:= \[Alpha]1=Transpose[{Range[0,itt],J[\[Theta]0,\[Theta]1]}];
In[20]:= \[Alpha]2=Transpose[{Range[0,itt],J[\[Theta]0,\[Theta]1]}];
In[26]:= \[Alpha]3=Transpose[{Range[0,itt],J[\[Theta]0,\[Theta]1]}];
In[32]:= \[Alpha]4=Transpose[{Range[0,itt],J[\[Theta]0,\[Theta]1]}];
In[38]:= \[Alpha]5=Transpose[{Range[0,itt],J[\[Theta]0,\[Theta]1]}];
```

Graph with ListLinePlot and visualize the learning curve for different alphas (see Figure 8-3). When changing the alpha value, check how the adjusted line changes.

```
In[39]:= ListLinePlot[{\[Alpha]1,\[Alpha]2,\[Alpha]3,\[Alpha]4,\[Alpha]5},
FrameLabel->{"Number of Iterations","Loss Function"},
Frame->True,PlotLabel->"Learning Curve",
PlotLegends->SwatchLegend[{Style["\[Alpha]=1",#],
Style["\[Alpha]=0.1",#],Style["\[Alpha]=0.01",#],
Style["\[Alpha]=0.001",#],Style["\[Alpha]=0.0001",#]},
LegendLabel->Style["Learning rate",White],
LegendFunction->(Framed[#,RoundingRadius->5,
Background->Gray]&)]&[White]
Out[39]=
```



**Figure 8-3.** The learning curve for the gradient descent algorithm

In the previous graph (see Figure 8-3), you can visualize the size of iterations concerning cost and how it varies depending on the alpha value. With a high learning rate, you can cover more ground at each step but risk exceeding the lowest point. To know whether the algorithm works, you must see that each new iteration's loss function is decreasing. The opposite case would indicate that the algorithm is not working correctly; this can be attributed to various factors, such as a code error or an incorrect learning rate value. As the graph shows, adequate alpha values correspond to small values between a scale of 1 to  $10^{-4}$ . It is not necessary to use these exact values; you can use values within this range. Depending on the form of the data, the algorithm may or may not converge with different alpha values as the same for the iteration steps. If you choose minimal alpha values, the algorithm can take a long time to converge, as you can see for alpha values  $10^{-3}$  or  $10^{-4}$ .

## Linear Regression

Despite being able to build the algorithms to perform a linear regression, the Wolfram Language has a specialized function for machine learning. In the case of linear regression problems, there is the Predict function. The Predict function can also work with different algorithms, not only regression task algorithms.

## Predict Function

The Predict function helps you predict values by creating a predictor function using the training data. It also allows you to choose different learning algorithms, the purpose of which is to predict a numerical, visual, categorical value or a combination. The methods to choose from are decision tree, gradient boosted tree, linear regression, neural network, nearest neighbors, random forest, and Gaussian process. Each method has options within it; the options vary depending on the algorithm chosen to train the predictor function. Let's look at the linear regression method. The input data for Predict can be in the form of a list of rules, associations, or a dataset.

## Boston Dataset

Let's look at the first example of loading the Boston Homes data from the Wolfram Data Repository (see Figure 8-4). This dataset contains information about housing in the Boston, Massachusetts, area. For more in-depth information, refer to the article “Hedonic Housing Prices and the Demand for Clean Air,” by David Harrison and Daniel Rubinfeld, in the *Journal of Environmental Economics and Management* (1978; 5[1], 81–102. [https://doi.org/10.1016/0095-0696\(78\)90006-2](https://doi.org/10.1016/0095-0696(78)90006-2)) or *Regression Diagnostics: Identifying Influential Data and Sources of Collinearity: 546* by David Belsley, Edwin Kuh, and Roy Welsch, (Wiley-Interscience, 2013).

```
In[40]:= bstn=ResourceData[ResourceObject["Sample Data: Boston Homes"]]
Out[40]=
```

CRIM	ZN	INDUS	CHAS	NOX	RM	AGE	DIS	RAD	TAX
0.00632	18	2.31	tract does not bound Charles river	0.538 ppm	6.575	65.2	4.09	1	296
0.02731	0	7.07	tract does not bound Charles river	0.469 ppm	6.421	78.9	4.9671	2	242
0.02729	0	7.07	tract does not bound Charles river	0.469 ppm	7.185	61.1	4.9671	2	242
0.03237	0	2.18	tract does not bound Charles river	0.458 ppm	6.998	45.8	6.0622	3	222
0.06905	0	2.18	tract does not bound Charles river	0.458 ppm	7.147	54.2	6.0622	3	222
0.02985	0	2.18	tract does not bound Charles river	0.458 ppm	6.43	58.7	6.0622	3	222
0.08829	12.5	7.87	tract does not bound Charles river	0.524 ppm	6.012	66.6	5.5605	5	311
0.14455	12.5	7.87	tract does not bound Charles river	0.524 ppm	6.172	96.1	5.9505	5	311
0.21124	12.5	7.87	tract does not bound Charles river	0.524 ppm	5.631	100	6.0821	5	311
0.17004	12.5	7.87	tract does not bound Charles river	0.524 ppm	6.004	85.9	6.5921	5	311
0.22489	12.5	7.87	tract does not bound Charles river	0.524 ppm	6.377	94.3	6.3467	5	311
0.11747	12.5	7.87	tract does not bound Charles river	0.524 ppm	6.009	82.9	6.2267	5	311
0.09378	12.5	7.87	tract does not bound Charles river	0.524 ppm	5.889	39	5.4509	5	311
0.62976	0	8.14	tract does not bound Charles river	0.538 ppm	5.949	61.8	4.7075	4	307
0.63796	0	8.14	tract does not bound Charles river	0.538 ppm	6.096	84.5	4.4619	4	307
0.62739	0	8.14	tract does not bound Charles river	0.538 ppm	5.834	56.5	4.4986	4	307
1.05393	0	8.14	tract does not bound Charles river	0.538 ppm	5.935	29.3	4.4986	4	307
0.7842	0	8.14	tract does not bound Charles river	0.538 ppm	5.99	81.7	4.2579	4	307
0.80271	0	8.14	tract does not bound Charles river	0.538 ppm	5.456	36.6	3.7965	4	307
0.7258	0	8.14	tract does not bound Charles river	0.538 ppm	5.727	69.5	3.7965	4	307

**Figure 8-4.** Boston Homes price dataset

Try using the scroll bars to have a complete view of the dataset. Let's look at the descriptions of the columns and show them in TableForm.



```

In[41]:= ResourceData[ResourceObject["Sample Data: Boston
Homes"], "ColumnDescriptions"]//TableForm
Out[41]//TableForm= Per capita crime rate by town
Proportion of residential land zoned for lots over 25000 square feet
Proportion of non-retail business acres per town
Charles River dummy variable (1 if tract bounds river, 0 otherwise)
Nitrogen oxide concentration (parts per 10 million)
Average number of rooms per dwelling
Proportion of owner-occupied units built prior to 1940
Weighted mean of distances to five Boston employment centers
Index of accessibility to radial highways
Full-value property-tax rater per $10000
Pupil-teacher ratio by town
1000(Bk-0.63)^2 where Bk is the proportion of Black or African-American
residents by town
Lower status of the population (percent)
Median value of owner-occupied homes in $1000s

```

## Model Creation

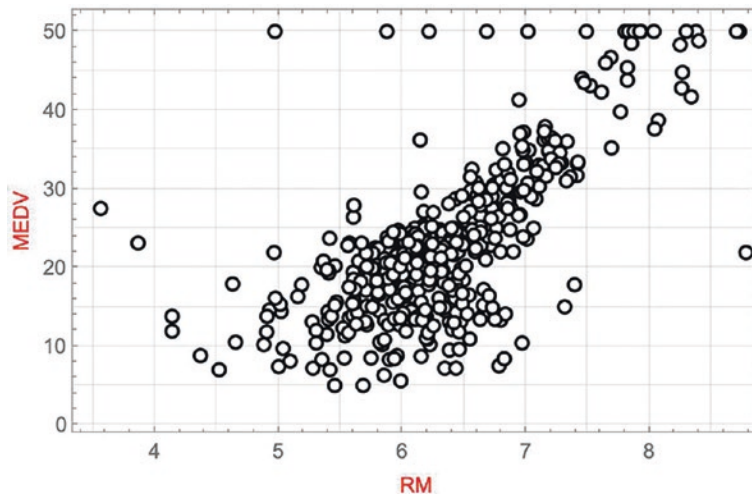
You create a model capable of predicting housing prices in the Boston area through the number of rooms in the dwelling. To achieve this, the columns of interest correspond to RM (average number of rooms per dwelling) and MEDV (median value of owner-occupied homes) since you want to find out if there is a linear relationship between the number of rooms and the price of the house. Applying some common sense, the houses with the most significant number of rooms are more extensive and, therefore, can store more people, increasing the price.

Look at the MEDV and RM scatter plots in Figures [8-5](#).

```

In[42]:= MEDVvsRM=Transpose[{Normal[bsn[All,"RM"]],Normal[bsn[All,"
MEDV"]]}];
ListPlot[MEDVvsRM,PlotMarkers->"OpenMarkers",Frame->True,FrameLabel->
{Style["RM",Red],Style["MEDV",Red]},GridLines->All,PlotStyle->
Black,ImageSize->Medium]
Out[43]=

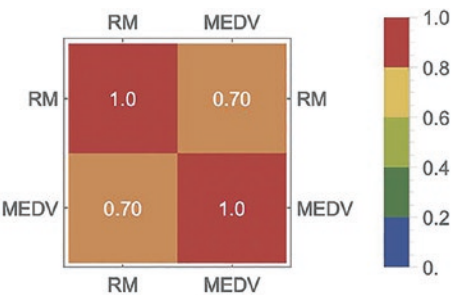
```



**Figure 8-5.** 2D scatter plot of MEDV vs. RM

As seen in Figure 8-5, the house price increases as the average number of rooms increases. This suggests that there is a direct proportional relationship between these two variables. Given what is seen in the graph, let's know the correlation value between these variables. You show this through a correlation matrix by first computing the correlation of the values, assigning the ticks' names, and plotting it with MatrixPlot (see Figure 8-6).

```
In[44]:=correlat=SetPrecision[Correlation[Transpose[{Normal[bsn[All,"RM"]],
Normal[bsn[All,"MEDV"]]}],2];
xTicks={{1,"RM"},{2,"MEDV"},{1,"RM"},{2,"MEDV"}};
yTicks={{1,"RM"},{2,"MEDV"},{1,"RM"},{2,"MEDV"}};
postionsValues={Text[#1,{0.5,1.5}],Text[#1,{1.5,0.5}],Text[#2,{1.5,1.5}],
Text[#2,{0.5,0.5}]}&[correlat[[1,1]],correlat[[1,2]]];
MatrixPlot[correlat,ColorFunction->"DarkRainbow",FrameTicks->{ xTicks,
yTicks,xTicks,yTicks},Epilog->{White,postionsValues},PlotLegends->
BarLegend[{"DarkRainbow",{0,1}},4],ImageSize->180]
Out[48]=
```



**Figure 8-6.** A matrix plot combined with a correlation matrix

By observing the matrix plot (see Figure 8-6), it can be concluded that there is an excellent linear relationship between RM and MEDV.

Let’s now shuffle the dataset randomly and establish a list of rules with Thread because the data to be entered in the predictor function must be as follows: {x → y}—in other terms, input, and target value.

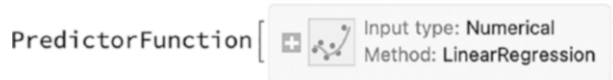
```
In[49]:= newData = RandomSample[Thread[Normal[bsn[All, "RM"]]] ->
Normal[bsn[All, "MEDV"]]]];
```

Once randomly sampled, you select the first 354 elements (70%); this is the training set, and the remaining 152 (30%) is the test set. When splitting, common ratios include 70/30 (training/testing), 80/20, and 60/40. Where the training set is used to train the model and usually the majority of the data. The remaining portion, the test set, is an independent dataset to assess the model performance on unseen data. The choice depends on factors like the size of the dataset and the detailed conditions of the machine-learning task you want to do.

```
In[50]:= {training, test} = {newData[;; 354]], newData[[355 ;;]]];
```

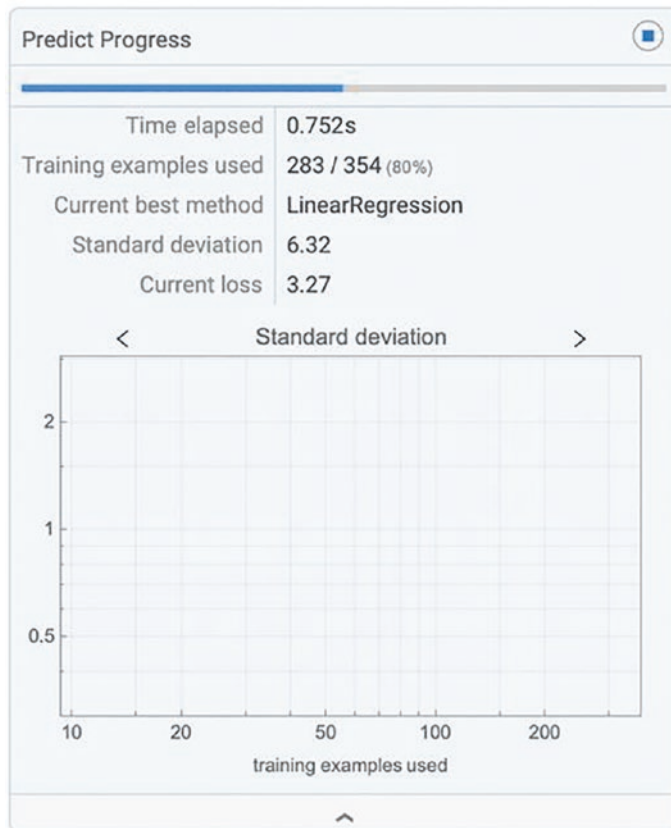
You train the model, a predictor for the average values of owner-occupied homes (MEDV) as a target. As a method, you choose linear regression. When training a model, specification of the option of training report includes Panel (dynamical updating of the Panel), Print (periodic information including time, training example, best method, current loss), ProgressIndicator (simple progress bar), SimplePanel (dynamic update panel with no plots), and None. Panel is the default option (see Figure 8-7).

```
In[51]:= pF=Predict[training,Method->"LinearRegression",TrainingProgress
Reporting->"Panel"]
Out[51]=
```



**Figure 8-7.** *PredictorFunction* object of the trained model

When entering the code, depending on the option added to `TrainingProgressReporting`, a progress bar and panel report should appear (see Figure 8-8). The time of the panel displayed depends on the training time of the model. To set a specific time for the training, add `TimeGoal` as an option, which specifies how long the training should last for the model. Time values are seconds of CPU time—that is, the number with no units. With units of time (seconds, minutes, and hours), the use of `Quantity` command is needed, like `TimeGoal ► Quantity[“time magnitude,” #] & / @ {“Second,” “Minute,” “Hour”}`.



**Figure 8-8.** *Progress report of the PredictorFunction*

Let’s go back to the model. Figure 8-7 shows that the return object is a predictor function (try using Head to verify it). When assigning a name to the predictor function, additional information about the model can be obtained; the command Information is used (see Figure 8-9). The information works for every other expression, not just for machine learning purposes.



**Figure 8-9.** Information report of the trained model

---

**Note** If you want fixed results involving random data, you need to set the seed before every random operation; this ensures consistent outputs.

---

```
In[52]:= Information[pF]
Out[52]=
```

The information panel in Figure 8-9 includes data type, root mean squared (StandardDeviation), method, batch evaluation speed, loss, model memory, number of examples for training, and training time. The graphics at the bottom of the panel are for standard deviation, model learning curve, and learning curve for the other algorithms. Hovering the cursor pointer over the numerical parameters shows the confidence intervals and units. If the method's name is correct, it shows the parameters of the linear regression method. Since you did not select a specific optimization algorithm within the LinearRegression method, Mathematica tries to search through the algorithms for the best one (this can be viewed in the learning curve for all algorithms). You see how to access these options further down the line.

---

**Note** Every method used in the predict function has options and suboptions; for full customization, use the Wolfram Language Documentation Center.

---

Table 8-1 shows the standard options that can be used for model training, as well as their definition and possible values for the training process of a PredictorFunction.

**Table 8-1.** *Most Common Options for Predict Function*

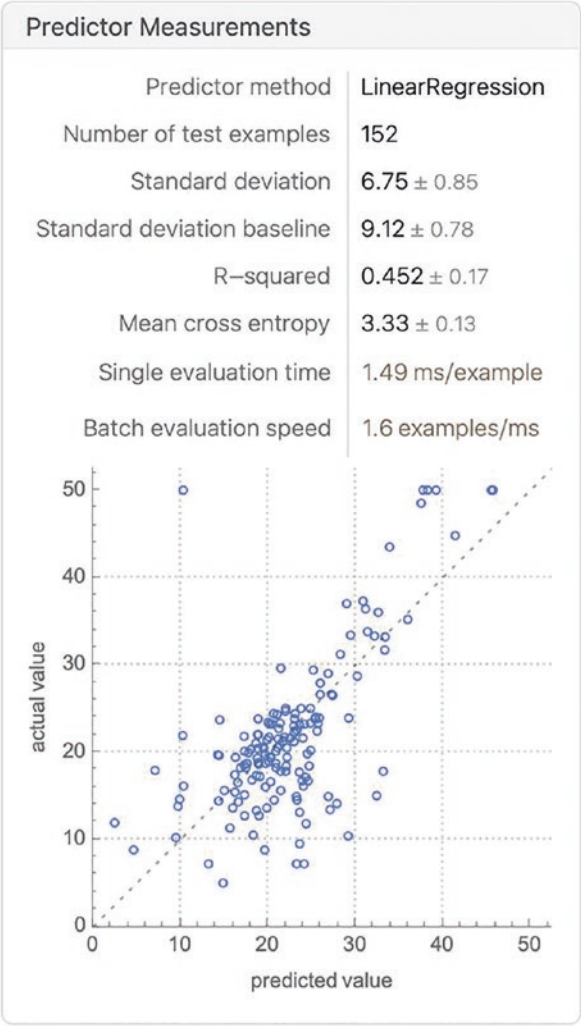
Option	Definition
Method	AlgorithmPossible values: DecisionTree, GradientBoostedTrees, LinearRegression, NearestNeighbors, RandomForest and GaussianProcess
PerformanceGoal	Performance optimizationPossible values: DirectTraining, Memory, Quality, Speed, TrainingSpeed, Automatic Combination of values supported (PerformanceGoal -> {val1, val2})
RandomSeeding	Seed for the pseudorandom number generatorPossible values: Automatic, "custom seed," Inherited (random seed used in previous computations)
TargetDevice	Specifies a device to perform the training or test processPossible values: CPU or GPU. If a GPU is installed, the automatic target device is the GPU.
TimeGoal	Time spent on the training process
TrainingProgressReporting	Progress reportPossible values: Panel, Print, ProgressIndicator, SimplePanel, None

---

# Model Measurements

Once the model is built, you must observe and analyze the performance of the predictor function in the test set. To carry out this, you must do it within the PredictorMeasurments command. The predictor function goes in the argument (see Figure 8-10), followed by the test set and the property or properties to add. Since the latest version, the final model features predictions are presented instead of just the model of the PredictorMeasurements object.

```
In[53]:= pRM=PredictorMeasurements[pF,test]
Out[53]=
```



**Figure 8-10.** *PredictorMeasurements* object of the tested model

The returned object is called `PredictorMeasurementsObject`. You can add the properties from the `PredictorMeasurements` command. You can assign a variable to the object to access it more simply. Since the new version of 13, the report is given in the output, so the model report of the test set is suppressed as it returns the same as in Figure 8-10.

```
In[54]:= pRM["Report"];
```

The report in Figure 8-10 shows different parameters, such as the standard deviation and mean cross-entropy. It shows a graph of the model's fit and the current and predicted values. The model is suitable for most cases, except that some outliers still affect performance.

To better understand the precision of the model, let's look at the root mean squared error (RMSE) and `RSquared` (coefficient of determination) shown in Figure 8-11. To display the associated uncertainties, use the option `ComputeUncertainty` with `True` value.

```
In[57]:=Dataset[AssociationMap[pRM[#,ComputeUncertainty-> True]&, {"Standard
Deviation","RSquared"}]]
Out[57]=
```

StandardDeviation	6.8 ± 0.8
RSquared	0.45 ± 0.17

**Figure 8-11.** *Standard deviation and r-squared values of the linear model*

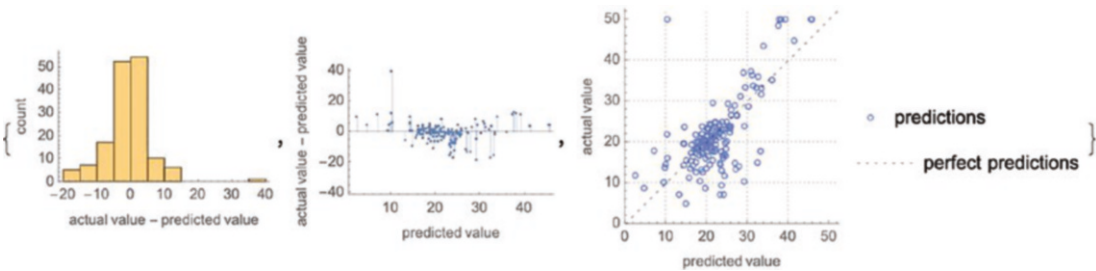
This gives you a slightly high RMSE value, not an excellent r-squared value. Remember that the r-squared value indicates how good the model is for making predictions. These two values indicate that although there may be a linear relationship between the number of rooms and prices, a linear regression does not necessarily explain this. These observations are also consistent, remembering that you obtained a correlation value of 0.7.



## Model Assessment

The graphs made within the model are the model graph and the target variable (ComparisonPlot). To check the distribution of the variance, use the ResidualHistogram function, and to check the residual plot, use ResidualPlot. These are shown in Figure 8-12.

```
In[58]:=pRM[#]&@{"ResidualHistogram", "ResidualPlot", "ComparisonPlot"} /.
plot_Graphics:>Show[plot,ImageSize->Small]
Out[58]=
```



**Figure 8-12.** *ResidualHistogram, ResidualPlot, and ComparisonPlot*

You write Properties as an argument to find out all the properties of the Predictor Measurements object. These properties can vary between methods.

```
In[59]:= pRM["Properties"]
Out[59]={BatchEvaluationTime,BestPredictedExamples,ComparisonPlot,
EvaluationTime,Examples,FractionVarianceUnexplained,GeometricMeanProbabili
tyDensity,ICEPlots,LeastCertainExamples,Likelihood,LogLikelihood,MeanCross
Entropy,MeanDeviation,MeanSquare,MostCertainExamples,Perplexity,PredictorFu
nction,ProbabilityDensities,ProbabilityDensityHistogram,Properties,Rejectio
nRate,Report,ResidualHistogram,ResidualPlot,Residuals,RSquared,SHAPPlots,SH
APValues,StandardDeviation,StandardDeviationBaseline,Totalsquare,WorstPredi
ctedExamples}
```

If you are not satisfied with the chosen methods or hyperparameters, retraining the model can be done by configuring the new values for the hyperparameters. You access the values of the current method with the help of the Information command and add the properties of Method (shows you the Method used to train the model), method description (description of the Method used), and MethodOption (method options).

```
In[60]:= Information[pF,"MethodOption"]
Out[60]=Method->{LinearRegression,L1Regularization->0,L2Regularization->
1.*10^-6,OptimizationMethod->NormalEquation}
```

You see terms such as L1Regularization, L2Regularization, and OptimizationMethod. The first two terms are associated with regularization methods, and L1 refers to the Lasso regression name and L2 to the Ridge regression name. Regularization is used to minimize the complexity of the model and reduce the variation; it also improves the precision of the model, solving overfitting problems. This is accomplished by adding a penalty to the loss function; this penalty is added to the sum of the absolute value of the coefficient  $\lambda_1 * \sum_{i=0}^N |\theta_i|$ , whereas for L2, it is given by the expression  $(\lambda_2 / 2) * \sum_{i=0}^N \theta_i^2$ , where the function to minimize is the loss function  $(1 / 2) \sum_{i=0}^N (y_i - f(\theta, x_i))^2$ . For more mathematical depth, refer *Artificial Intelligence: A Modern Approach* by Stuart Russell and Peter Norvig (Prentice Hall, 2010) and *An Introduction to Statistical Learning: With Applications in R* by Gareth James, Trevor Hastie, Robert Tibshirani, and Daniela Witten (Springer, 2017). The third term is which optimization method you want to choose; the existing methods are NormalEquation, StochasticGradientDescent, and OrthantWiseNewton. That said, it must be emphasized that using the vector of coefficients with the L1 and L2 standards is known as an Elastic Net regression model. Elastic Net might be used when there is a correlation in the parameters. For more theory, reference *The Elements of Statistical Learning: Data Mining, Inference, and Prediction* by Trevor Hastie, Robert Tibshirani, and Jerome Friedman (Springer, 2009).

## Retraining Model Hyperparameters

As discussed later, let's retrain the model but with the values of L1  $\rightarrow$  12, L2  $\rightarrow$  100 and the optimization algorithm OptimizationMethod  $\rightarrow$  StochasticGradientDescent, TrainingProgressReporting  $\rightarrow$  None, PerformanceGoal  $\rightarrow$  "Quality," RandomSeeding  $\rightarrow$  10000, TargetDevice  $\rightarrow$  "CPU."

```
In[61]:= pF2 = Predict[training, Method -> {"LinearRegression",
"L1Regularization" -> 12, "L2Regularization" -> 100, "OptimizationMethod"
-> Automatic}, TrainingProgressReporting -> None, PerformanceGoal ->
"Quality", RandomSeeding -> 10000, TargetDevice -> "CPU"];
```

To see the properties related to an example, type properties after the input data for the PredictorFunction—for instance, PF2[“example,” “Properties”]. Let’s compare the new model’s performance by showing the graphs and metrics like before (see Figures 8-13 and 8-14).

**Note** Standard deviation refers to the root mean square of the residuals, root mean square error (RMSE).

```
In[62]:= pRM2=PredictorMeasurements[pF2,test];
pRM2[#]&/@{"ResidualHistogram","ResidualPlot","ComparisonPlot"}/.
plot_Graphics:>Show[plot,ImageSize->Small]
Dataset[AssociationMap[pRM2[#,ComputeUncertainty->True]&,{"StandardDeviation",
"RSquared"}]]]
Out[63]=
Out[64]=
```

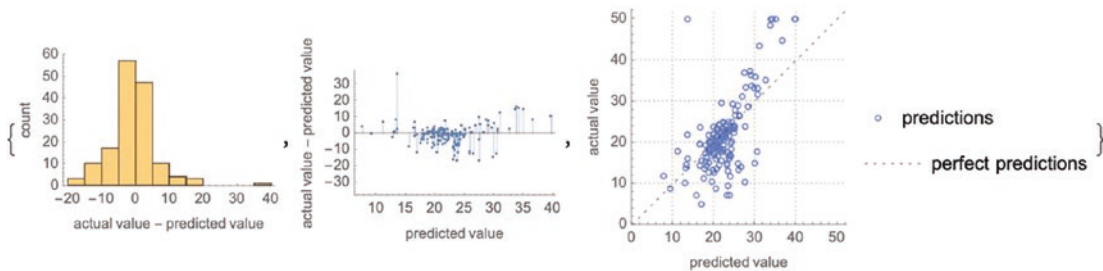


Figure 8-13. Plots of the retrained model

StandardDeviation	6.8 ± 0.7
RSquared	0.44 ± 0.15

Figure 8-14. New values for standard deviation and r-squared

Observing the graphs and data, you see the model merely decreases to a certain degree; this agrees with the new  $r$ -squared value. However, it is still a poor model for making future predictions. The poor performance may be due to the optimization choice, the L1 and L2 parameters. Try to explore different L1 and L2 values for potential improvement.

## Logistic Regression

Logistic regression is a technique commonly used in statistics but also used within machine learning. The logistic regression works considering that the values of the response variable only take two values, 0 and 1; this can also be interpreted as a false or true condition. It is a binary classifier that uses a function to predict the probability of whether or not a condition is met, depending on how the model is constructed. Usually, this model type is used for classification since it can provide you with probabilities and classifications since the values of the logistic regression oscillate between two values. In logistic regression, the target variable is a binary variable that contains encoded data. For more information, refer to *Introduction to Data Science: A Python Approach to Concepts, Techniques and Applications* by Laura Igual, Santi Seguí, Jordi Vitrià, Eloi Puertas, Petia Radeva, Oriol Pujol, Sergio Escalera, Francesc Dantí, and Lluís Garrido (Springer, 2017).

## Titanic Dataset

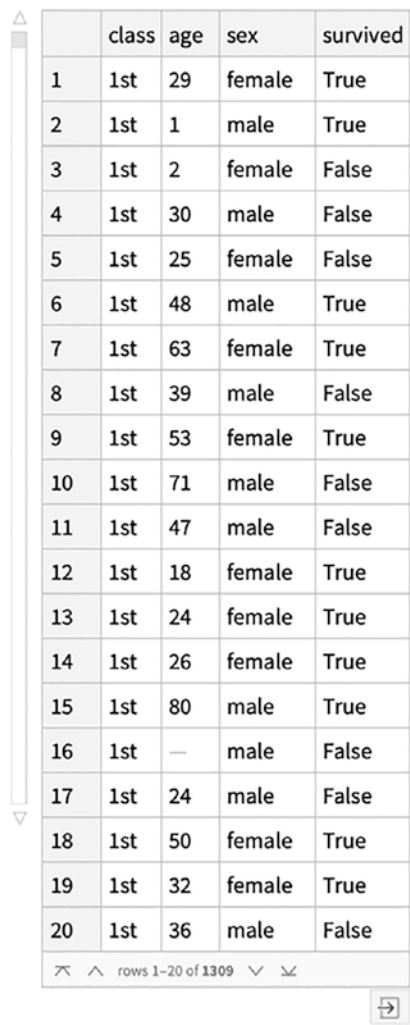
For the following example, you use the Titanic dataset, which is a dataset that describes the survival status of the passengers. The variables used are class, age, sex, and survival condition. You load the data directly as a dataset (see Figure 8-15) from the ExampleData and enumerate the rows of the dataset.

---

**Note** This section is constructed using Query language so the reader can understand how to use it more deeply inside datasets.

---

```
In[65]:= titanic=Query[AssociationThread[Range[Length@#]->Range[Length@#]]]
[ExampleData[{"Dataset","Titanic"}]]&[ExampleData[{"Dataset","Titanic"}]]
Out[65]=
```



	class	age	sex	survived
1	1st	29	female	True
2	1st	1	male	True
3	1st	2	female	False
4	1st	30	male	False
5	1st	25	female	False
6	1st	48	male	True
7	1st	63	female	True
8	1st	39	male	False
9	1st	53	female	True
10	1st	71	male	False
11	1st	47	male	False
12	1st	18	female	True
13	1st	24	female	True
14	1st	26	female	True
15	1st	80	male	True
16	1st	—	male	False
17	1st	24	male	False
18	1st	50	female	True
19	1st	32	female	True
20	1st	36	male	False

**Figure 8-15.** *New values for Standard deviation and r-squared*

Let’s look at the dimensions of the data using the Dimensions command.

```
In[66]:= Dimensions@titanic
Out[66]= {1309,4}
```

Interpreting the result, you see that the dataset comprises 1309 rows and four columns. The dataset has four columns classified by class, age, sex, and survived status. Using the space bar shows that some elements do not register data entry. To see which columns contain missing data, execute the following code by counting the components corresponding to the pattern missing in each column.


```
In[67]:=Query[Count[_Missing],#]@titanic&/@{"class","age","sex","survived"}
Out[67]= {0,263,0,0}
```

This shows 263 missing values within the age column and zero for the others. Let's remove the rows that contain this missing data. First, you extract the row numbers from the missing data by selecting the elements from the age column equal to missing and then extracting the row IDs.

```
In[68]:= Query[Select[#age==Missing[ ]&]][titanic];
Normal@Keys@%
Out[68]={16,38,41,47,60,70,71,75,81,107,108,109,119,122,126,135,148,153,
158,167,177,180,185,197,205,220,224,236,238,242,255,257,270,278,284,294,
298,319,321,364,383,385,411,470,474,478,484,492,496,525,529,532,582,596,
598,673,681,682,683,706,707,757,758,768,769,776,790,796,799,801,802,803,
805,806,809,813,814,816,817,820,836,843,844,853,855,857,859,866,872,873,
875,877,880,883,887,888,901,902,903,904,919,921,922,923,924,927,928,929,
930,931,932,941,943,945,946,947,949,955,956,957,958,959,962,963,972,974,
977,983,984,985,988,989,990,992,994,995,998,999,1000,1001,1002,1003,1004,
1005,1006,1007,1010,1013,1014,1015,1017,1019,1023,1024,1028,1029,1030,1031,
1033,1034,1035,1036,1037,1038,1039,1040,1042,1043,1044,1045,1053,1054,1055,
1056,1070,1071,1072,1073,1074,1075,1077,1078,1079,1081,1082,1086,1096,1110,
1115,1116,1117,1122,1123,1124,1125,1129,1133,1136,1137,1138,1139,1150,1151,
1152,1155,1156,1160,1163,1164,1165,1167,1168,1169,1171,1173,1174,1175,1176,
1177,1178,1179,1180,1181,1185,1186,1187,1194,1195,1196,1198,1199,1200,1201,
1203,1213,1214,1215,1216,1217,1220,1222,1242,1243,1244,1246,1247,1248,
1250,1251,1254,1256,1263,1269,1283,1284,1285,1292,1293,1294,1298,1303,
1304,1306}
```

These numbers represent the rows containing the age column's missing data. You use the `DeleteMissing` command to eliminate them, considering there is missing data at level 1. The final dataset is seen in (see Figure 8-16).

```
In[69]:= titanic>DeleteMissing[titanic,1,1]
Out[69]=
```



	class	age	sex	survived
1	1st	29	female	True
2	1st	1	male	True
3	1st	2	female	False
4	1st	30	male	False
5	1st	25	female	False
6	1st	48	male	True
7	1st	63	female	True
8	1st	39	male	False
9	1st	53	female	True
10	1st	71	male	False
11	1st	47	male	False
12	1st	18	female	True
13	1st	24	female	True
14	1st	26	female	True
15	1st	80	male	True
17	1st	24	male	False
18	1st	50	female	True
19	1st	32	female	True
20	1st	36	male	False
21	1st	37	male	True

rows 1-20 of 1046

**Figure 8-16.** *Titanic dataset without missing values*

To corroborate that there is no missing data, you could apply the same code with counts or by looking at the keys of the removed rows, for example.

```
In[70]:= titanic[Key[16]]
Out[70]= Missing[KeyAbsent,Key[16]]
```

This means that there is no content associated with key number 16. If you want to check all keys, use the row list of the missing data.

## Data Exploration

Once you have removed the missing data, you can count the elements of each class, sex, and survival status (see Figure 8-17).

```
In[71]:= Dataset@<|"Class" -> Query[Counts, "class"]@titanic, "Sex"
-> Query[Counts, "sex"]@titanic, "Survival status" -> Query[Counts,
"survived"]@titanic|>
Out[71]=
```

Class	1st	284
	2nd	261
	3rd	501
Sex	female	388
	male	658
Survival status	True	427
	False	619

**Figure 8-17.** Basic elements count for class, sex, and survival status

After eliminating the rows with the missing elements, the dataset consists of 284 elements in the first class, 261 in the second class, and 501 in the third class (see Figure 8-18). Also, note that more than half of the registered passengers were male and that there were more deaths than survivors. It is possible to verify this graphically by showing the percentages. The same approach is applied to the column's class and sex.

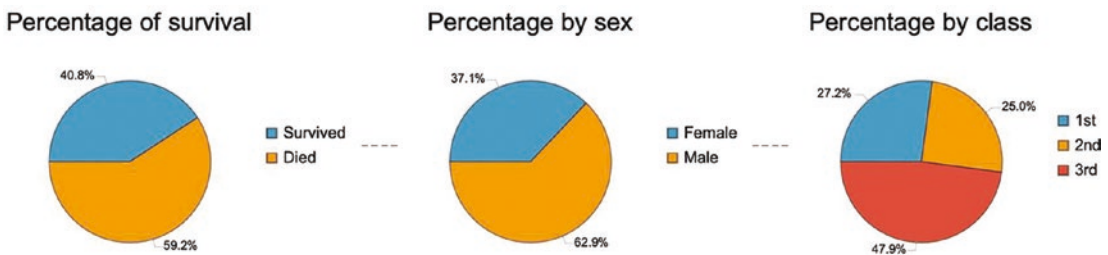
```
In[72]:= Row[{PieChart[{N@(#[[1]]/Total@#),N@(#[[2]]/Total@#)}&[Counts
[Query[All,"survived"]@titanic]], PlotLabel->Style["Percentage of
survival",#3,#4], ChartLegends-> {"Survived", "Died"}, ImageSize->#1,
ChartStyle->#2,LabelingFunction->(Placed[Row[{SetPrecision[100#,3],"%"}],
"RadialCallout"]&)],
```



```

PieChart[{N@#[[1]]/Total@#,N@#[[2]]/Total@#}&[Counts[Query[All,"sex"]
[titanic]]],PlotLabel->Style["Percentage by sex",#3,#4],ChartLegends->
{"Female","Male"},ImageSize->#1,ChartStyle->#2,LabelingFunction->(Placed
[Row[{SetPrecision[100#,3],"%"},"RadialCallout"]&)],
PieChart[{N@#[[1]]/Total@#,N@#[[2]]/Total@#,N@#[[3]]/Total@#}&
[Counts[Query[All,"class"][titanic]]],PlotLabel->Style["Percentage by
class",#3,#4],ChartLegends->{"1st","2nd","3rd"},ImageSize->
#1,ChartStyle->#2,LabelingFunction->(Placed[Row[{SetPrecision[100#,3],"%"}],
"RadialCallout"]&)],"----"&[200,{ColorData[97,20],ColorData[97,13],
ColorData[97,32]}],Black,20]
Out[72]=

```



**Figure 8-18.** Pie charts for class, sex, and survival status

This example looks at the survival status of *Titanic* passengers. It builds a model that classifies whether a given class, age, and sex survived and which did not. The features are class, age, and sex; the target is survival status. These variables are the features, which the model then uses to classify whether a class, age, and sex survived, which is the target variable. The dataset is divided into 80% training (837 elements) and 20% test (209 elements). To split the dataset, first do a random sampling; afterward, extract the keys of the IDs and create a new dataset divided by the train and test sets (see Figure 8-19).

```

In[73]:= BlockRandom[SeedRandom[8888];
RandomSample[titanic]];
Keys@Normal@Query[All][%];
{train,test}={%[[1;;837]],%[[838;;1046]]];
dataset=Query[<|"Train"->{Map[Key,train]},"Test"->{Map[Key,test]} |> ]
[titanic]
Out[77]=

```

		class	age	sex	survived
Train	410	2nd	36	male	False
	537	2nd	32	female	True
	874	3rd	42	male	False
	691	3rd	22	male	False
	1021	3rd	21	male	False
	852	3rd	45	female	True
	705	3rd	21	male	False
	743	3rd	45	male	True
	515	2nd	2	male	True
	658	3rd	1	female	True
	837 total				
Test	1227	3rd	19	male	False
	188	1st	16	female	True
	397	2nd	34	female	True
	944	3rd	37	female	False
	262	1st	35	male	True
	95	1st	4	male	True
	1080	3rd	22	female	True
	918	3rd	39	male	True
	517	2nd	37	male	False
	425	2nd	30	male	False
	209 total				

**Figure 8-19.** *Titanic dataset divided by train and test set*

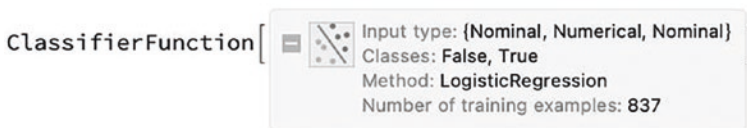
## Classify Function


The `Classify` command is another super function used in the Wolfram Language machine learning scheme. This function can be used in tasks that solve a classification problem. The data that this function accepts are numerical, textual, sound, and image data. This function's input data can be the same as the `Predict` function  $\{x \rightarrow y\}$ . However, entering data as a list of elements, an association of elements, or a dataset is also possible. In this case, you introduce it as a dataset.

In this case, you extract the data from the dataset format by specifying that the columns' input (class, age, sex) points to the target (survived). Now, let's build the classifier function (see Figure 8-20) with the following options: `Method`  $\rightarrow$  `{LogisticRegression, L1  $\rightarrow$  Automatic, L2  $\rightarrow$  Automatic}`. When choosing `Automatic`,

you let Mathematica choose the best combination of L1 and L2 parameters. For the `OptimizationMethod`, set the `StochasticGradientDescent` method. And for performance goal set `Quality`. Finally, you choose a seed with a value of 100,000 and the CPU unit as the target device. The optimization methods for the logistic regression are the limited memory Broyden-Fletcher-Goldfarb-Shanno algorithm, `StochasticGradientDescent`, and Newton method. These are for estimating the parameters of the logistic function. The rule construction is done from the data inside the dataset using the Query language.

```
In[78]:= cF = Classify[Flatten[Values[Normal[Query["Train", All,
All, {#class, #age, #sex} -> #survived &][dataset]]]], Method ->
{"LogisticRegression", "L1Regularization" -> Automatic,
"L2Regularization" -> Automatic, "OptimizationMethod" ->
"StochasticGradientDescent"}, PerformanceGoal -> "Quality", RandomSeeding
-> 100000, TargetDevice -> "CPU", TrainingProgressReporting -> None]
Out[78]=
```

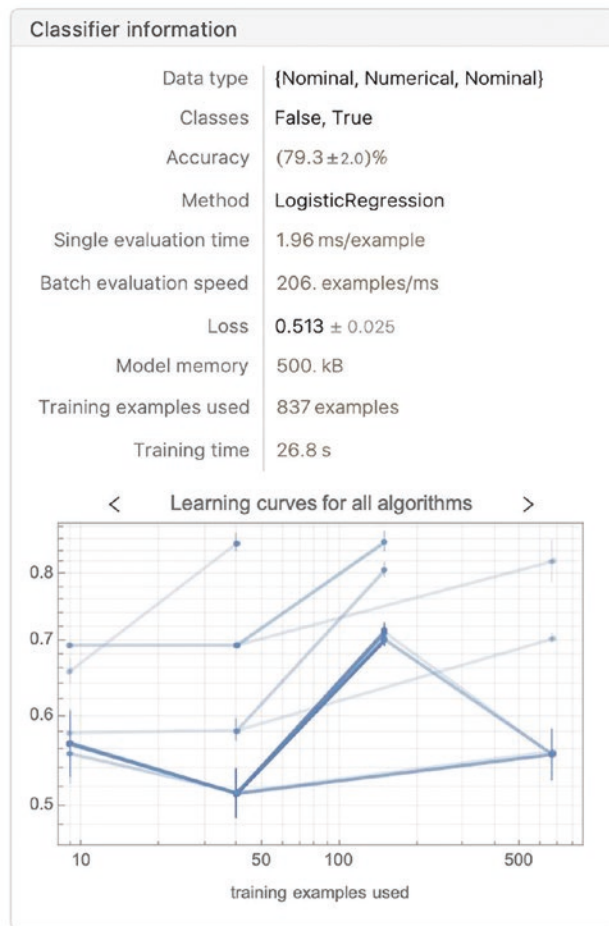


```
ClassifierFunction[  Input type: {Nominal, Numerical, Nominal}
Classes: False, True
Method: LogisticRegression
Number of training examples: 837 ]
```

**Figure 8-20.** *ClassifierFunction object*

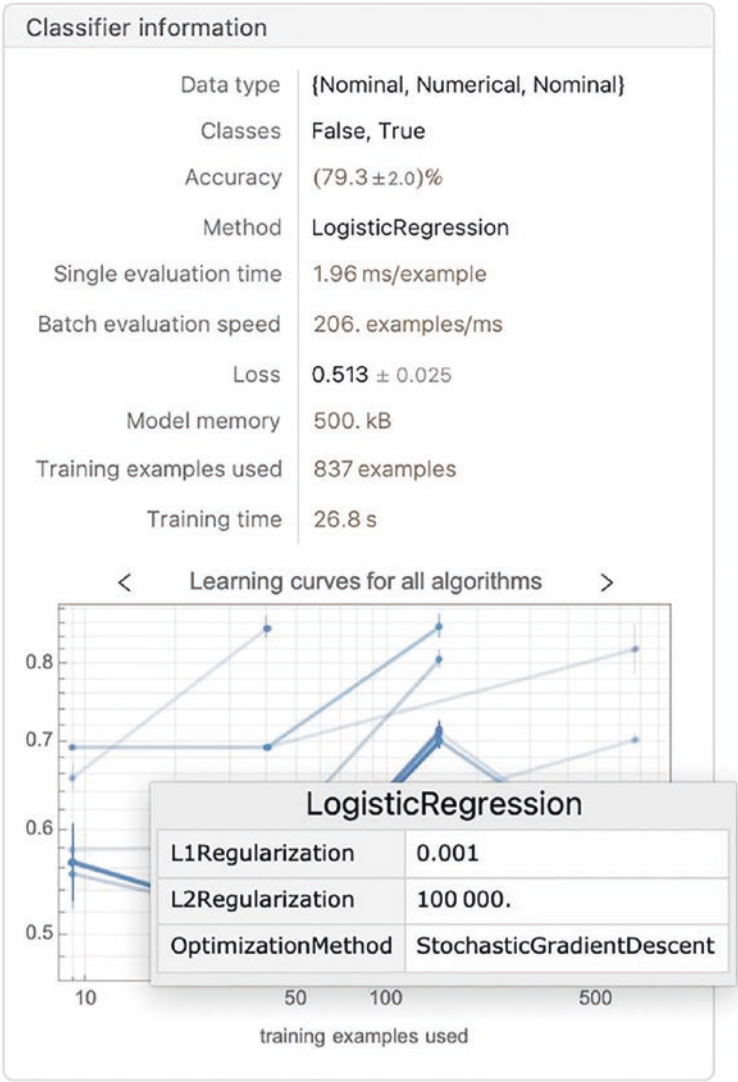
After training, like with the `Predict` function, the `Classify` function returns a classifier function object (see Figure 8-21) instead of a predictor function. Inspecting the classifier function, you can see the two input data types—nominal and numerical—and the classes, which are the survival status—true or false. The method used (logistic regression) and the number of examples (837). To obtain information on the model, use the `Information` command. Let's look at the model report.

```
In[79]:= Information[cF]
Out[79]=
```



**Figure 8-21.** Information about the trained classifier function

**Note** If you click the arrows above the graphs, three plots are shown: Learning curve, accuracy, and Learning curve for all algorithms. If you hover the pointer over the line of the last one, a tooltip appears with the corresponding parameters along with the method used, as shown in Figure 8-22.



**Figure 8-22.** Algorithm specifications tooltip from the method logistic regression

You see that the model’s accuracy is approximately 79%. You also observe by clicking the arrows of the plots that the learning curve and accuracy curve both experience variation at 500 training examples used. To access all the properties of the trained model, add Properties as an option in Information.

```
In[80]:= Information[cF,"Properties"]
Out[80]={AcceptanceThreshold,Accuracy,AnomalyDetector,BatchEvaluationSpeed,
```

```
BatchEvaluationTime, Calibrated, Classes, ClassNumber, ClassPriors, Evaluation
Time, ExampleNumber, FeatureExtractor, FeatureNames, FeatureNumber, FeatureTypes,
FunctionMemory, FunctionProperties, IndeterminateThreshold, LearningCurve, Max
TrainingMemory, MeanCrossEntropy, Method, MethodDescription, MethodOption, Method
Parameters, MissingSynthesizer, PerformanceGoal, Properties, TrainingClassPriors,
TrainingTime, UtilityFunction}
```

---

**Note** Depending on the method used, properties may vary.

---

Let's examine the probabilities for the data: class = 3rd, age = 23, and sex = male. Probability → name or number of class or TopProbabilities → number of most likely classes.

```
In[81]:= cF[{"3rd", 23, "male"}, {"Probability" ->
False, "TopProbabilities" -> 2}]
Out[81]= {0.676982, {False -> 0.676982, True -> 0.323018}}
```

The probabilities of the latter example show that the passenger's survival status may be more inclined to the False status.

To see the complete properties of a new classification, type the example followed by Properties. The properties included are Decision (best choice of class according to probabilities and its utility function) and Distribution (categorical distribution object). Probabilities of each class are displayed as associations: ExpectedUtilities (expected probabilities), LogProbabilities (natural logarithm probabilities), Probabilities (all classes), and TopProbabilities (most likely class). This is displayed in the following dataset (see Figure 8-23).

```
In[82]:= Dataset@
AssociationMap[cF[{"3rd", 23, "male"}, #]
&, {"Decision", "Distribution", "ExpectedUtilities", "LogProbabilities",
"Probabilities", "TopProbabilities"}]
Out[82]=
```

Decision	False
Distribution	CategoricalDistribution[ <div>Input type: Scalar Categories: False True</div> ]
ExpectedUtilities	<  False → 0.676982, True → 0.323018, Indeterminate → 0.  >
LogProbabilities	<  False → -0.39011, True → -1.13005  >
Probabilities	<  False → 0.676982, True → 0.323018  >
TopProbabilities	{False → 0.676982, True → 0.323018}

**Figure 8-23.** Properties for the classifier function of the trained model

---

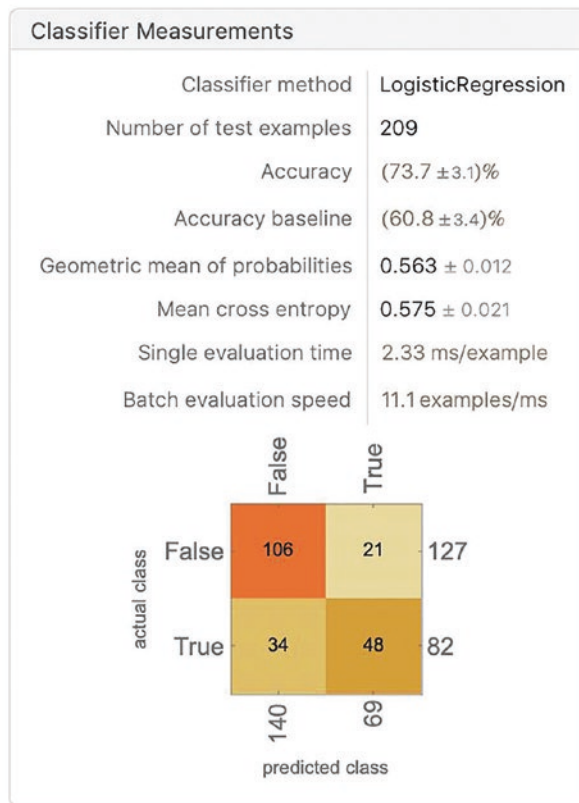
**Note** To check the logarithm result, use the Log command, Log[base, number].

---

## Testing the Model

You now test the model on the test data using the ClassifierMeasurements command, adding the function and the test set as arguments and the uncertainty computation. Like PredictionMeasurement, the output returned shows details about the model (see Figure 8-24).

```
In[83]:= cM = ClassifierMeasurements[cF,Flatten[Values[Normal[Query[
"Test", All, All, {#class, #age, #sex} -> #survived &][dataset]]]],
ComputeUncertainty -> True, RandomSeeding -> 8888]
Out[83]=
```



**Figure 8-24.** *ClassifierMeasurements* object of the classifier function

The object returned is called a `ClassifierMeasurementsObject` (see Figure 8-25), which is used to look for the properties of the `ClassifierFunction` after testing the test set. Just like with the linear regression model, the report of the test set is suppressed as it returns the same as in Figure 8-24.

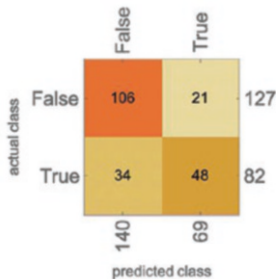
```
In[84]:=cM["Report"];
```



The report in Figure 8-24 shows information such as the number of test examples, the accuracy, and the accuracy baseline, among others. It also shows you the confusion matrix, which shows you the prediction results for the classification model, showing the number of correct and incorrect predictions; these being broken down by class, in this case, return either false or true, which gives you an idea of the errors the model is making and the type of error it is making. It shows you the true positives and true negatives and false positives and false negatives for each class.

Let’s look at the graph (confusion matrix) concretely (see Figure 8-25).

```
In[85]:= cM["ConfusionMatrixPlot"]
Out[85]=
```

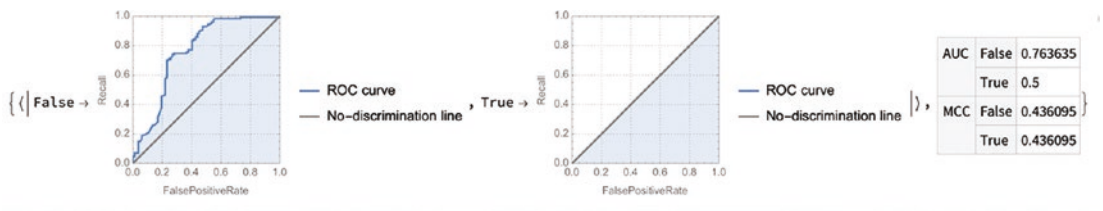


**Figure 8-25.** Confusion matrix plot of the tested model

To get the values of the confusion matrix, use CM[“ConfusionMatrix”] or class CM[“ConfusionFunction”].

Looking at the plot, you see that the model classified, starting from left to right at the top, 106 examples of false correctly classified, 21 examples of false as true, 34 examples of true as false, and 48 examples of true correctly. To better visualize the performance, look at each class’s ROC curves (see Figure 8-26), their respective values, and the Matthews correlation coefficient and AUC values.

```
In[86]:= {cM["ROCCurve"],Dataset@<|{"AUC"->cM["AreaUnderROCCurve"]},
{"MCC"->cM["MatthewsCorrelationCoefficient"]}|>}
Out[86]=
```



**Figure 8-26.** ROC curves for each class, along with AUC and MCC values

The two classes have different values in the AUC, but comparing the ROC curve; the class False has better classification than the True class. Let's look at which class has worse examples. You can show the less accurate results of the model, which has the highest entropy distribution and mean cross-entropy for each class.

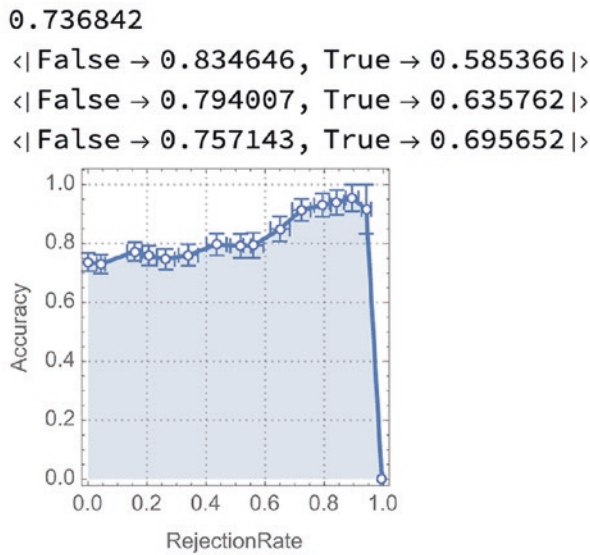
```
In[87]:= cM[{"LeastCertainExamples", "ClassMeanCrossEntropy"}]
Out[87]= {{{{1st,4,male}->True, {1st,19,male}->False, {1st,22,male}->
False, {1st,24,male}->False, {1st,25,male}->False, {1st,27,male}->
False, {1st,29,male}->False, {1st,30,male}->False, {1st,33,male}->False,
{1st,35,male}->True}, <|False->0.552204,True->0.611137|>}}
```

To get the values of the MCC coefficient, use the following properties: FalseDiscoveryRate, FalsePositiveRate, FalseNegativeRate (for each class), FalseNegativeExamples, FalseBegativeNumber (true negatives), FalsePositive and FalsePositiveNumber (true positive). These are shown in a short form here.

```
In[88]:= cM[#] & /@ {"FalseDiscoveryRate", "FalseNegativeRate",
"FalsePositiveRate"}
Out[88]= {<|False->0.242857,True->0.304348|>,<|False->0.165354,True->
0.414634|>,<|False->0.414634,True->0.165354|>}
```

Another way to see if the model behaves consistently in predictions is to look at key metric values like accuracy, recall, F1 score, precision, and the accuracy rejection plot (see Figure 8-27). Let's look at these metrics for the model.

```
In[89]:= cM[{"Accuracy", "Recall", "F1Score",
"Precision", "AccuracyRejectionPlot"}] // TableForm
Out[89]//TableForm=
```



**Figure 8-27.** *TableForm for the values of Accuracy, Recall, F1Score, Precision, and AccuracyRejectionPlot*

To see related metrics about the accuracy, type the following properties: Accuracy (number of correctly classified examples), AccuracyBaseline (accuracy of predicting the standard class), and AccuracyRejectionPlot (ARC plot, accuracy rejection curve). However, to find information about probability and the predicted class of the test set, use the following properties: DecisionUtilities (value of the utility function for every example in the test set), Probabilities (probabilities for every example in the test set), and ProbabilityHistogram (histogram of class probabilities). Let's look at how the probability behaves by plotting the probability of a passenger's survival status (see Figure 8-28), remembering that the false state means that a passenger did not survive, and True means that a passenger did survive.

```
In[90]:= plotClass[class1_, class2_, class3_, gender_, prob_,
frame_, ticks_,
imgSize_] := Plot[{cF[{class1, age, gender}, "Probability" ->
prob], cF[{class2, age, gender}, "Probability" -> prob], cF[{class3,
age, gender}, "Probability" -> prob]}], {age, 0, 90}, PlotLegends ->
{gender <> " in 1st class", gender <> " in 2nd class", gender
<> " in 3rd class"}, FrameLabel -> {Style["Age in years", Bold,
15], Style["Probability", Bold, 15]}, Frame -> frame, FrameTicks ->
```

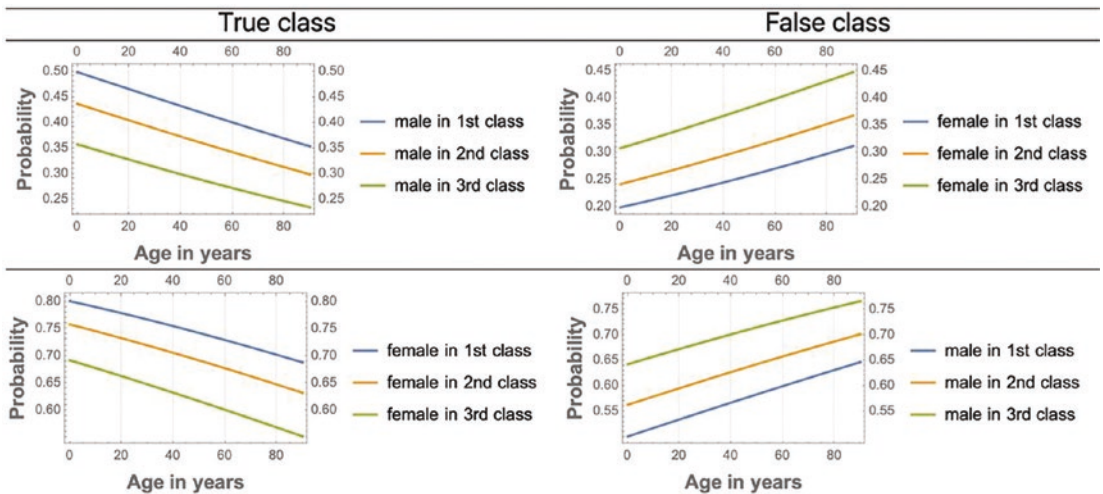
```

ticks, GridLines -> {{20, 40, 60, 80}}, ImageSize -> imgSize]

truPlot = {plotClass["1st", "2nd", "3rd", "male", True, True,
All, Medium], plotClass["1st", "2nd", "3rd", "female", True, True,
All, Medium]};
falsePlot = {plotClass["1st", "2nd", "3rd", "male", False, True,
All, Medium], plotClass["1st", "2nd", "3rd", "female", False, True,
All, Medium]};
headings = {Style["True class", Black, 20, FontFamily -> "Arial
Rounded MT"], Style["False class", Black, 20, FontFamily -> "Arial
Rounded MT"]};

Grid[{{headings[[1]], headings[[2]]}, {truPlot[[1]], falsePlot[[2]]},
{truPlot[[2]], falsePlot[[1]]}}, Alignment -> {{Center, Center}, {None,
None}}, Dividers -> {False, 1}]
Out[92]=

```



**Figure 8-28.** Probabilities of each class, depending on the class, age, and sex

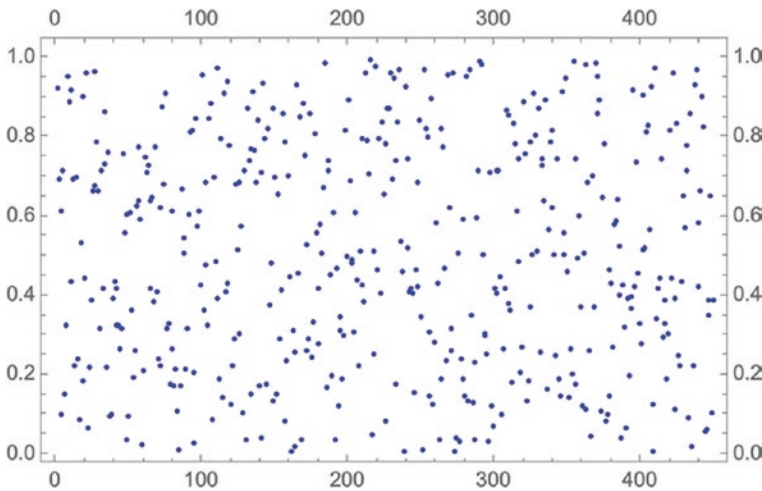
The graphs shown in Figure 8-30 clearly show that males' probability of survival decreases as age increases, even to hit values below 20% of chance, whether 1st, 2nd, or 3rd class. This is contrary to the probability of survival for females, where it starts with values above 80% of chance and decreases as age increases, too, hitting values above 50% for 1st class.

## Data Clustering

The data clustering method is unsupervised learning, as referenced by M. Emre Celebi, and Kemal Aydin in *Unsupervised Learning Algorithms* (Springer, 2018). It is generally used to find structures and characteristics of data clusters, where the points to be observed are divided into different groups by which they are compared based on unique characteristics.

The following example creates a bivariate data series and plot the list of points (see Figure 8-29). To find clusters, there is the Find Clusters command; this command makes a partition of the points according to their similarities.

```
In[93]:= BlockRandom[
SeedRandom[321];
rndPts=Table[{i,RandomReal[{0,1}]},{i,1,450}];]
ListPlot[rndPts,PlotRange->All,PlotStyle->Directive[Thick,Blue],Frame->
True,FrameTicks->All]
Out[93]=
```



**Figure 8-29.** 2D scatter plot of random data

## Clusters Identification

The FindClusters function is used to detect partitions within a set of data with similar characteristics. This function gathers the cluster elements into subgroups that the function finds. When you do not add options to the Find Clusters command,

Mathematica automatically sets the cluster identification parameters. Options for other machine learning methods can also be used for this command; for example, `PerformanceGoal`, `Method`, and `RandomSeeding`.

```
In[94]:= clusters=FindClusters[rndPts,PerformanceGoal->"Speed",Method-
>Automatic,DistanceFunction->Automatic,RandomSeeding->1234];
Short[clusters,1]
Out[95]//Short={{1,0.924416},{8,0.951038},<<162>>,{443,0.824999}},{<<1>>},
{<<1>>}}
```

Let's look at how many clusters were identified. You use the `Length` command; this way, you obtain the general form of the list.

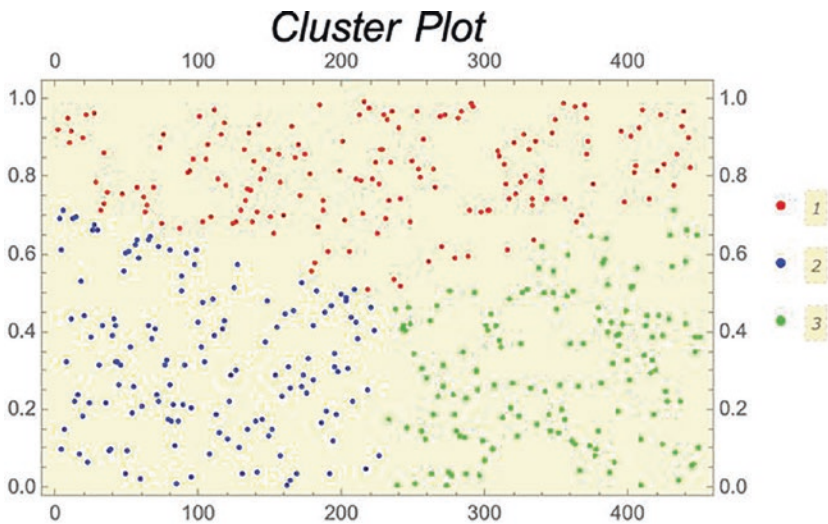
```
In[96]:= Length[clusters]
Out[96]= 3
```

You see that the result is three. This can be interpreted as follows: the list contains three elements (that is, three sublists), each list represents a cluster, and within each cluster, there is a sublist, which includes the points of each identified cluster. To determine how many elements are included in each cluster, use the `Map` command and apply the `Dimension` command at the specification level.

```
In[97]:= Map[Dimensions,clusters,1]
Out[97]= {{165,2},{143,2},{142,2}}
```

This tells you that the first cluster contains 165 elements, the second cluster contains 143 components, and the third cluster contains 142 elements; these are the same number of points you created earlier, totaling 450. Each cluster consists of a two-point coordinate system. The `FindClusters` command returns the points where it identifies the clusters. Figure 8-30 exhibits the plot of the clusters generated.

```
In[98]:= ListPlot[clusters,PlotStyle->{Red,Blue,Green},PlotLegends->
Automatic,Frame->True,FrameTicks->All,PlotLabel->Style["Cluster Plot",
Italic,20,Black],Prolog-> {LightYellow,Rectangle[Scaled[{0,0}],
Scaled[{1,1}]]}]
Out[98]=
```



**Figure 8-30.** 2D scatter plot of the three clusters identified

Find Clusters automatically colors the clusters. To explicitly establish the number of clusters to search, you add the desired number as the second argument—that is, in the form FindCluster [“points,” “a number of clusters”]. In the previous example, you set the method option to automatic. The different methods for finding the clusters are shown here. Agglomerate (which is the algorithm of single linkage clustering), density-based spatial clustering of applications with noise (DBSCAN), NeighborhoodContraction (nearest-neighbor chain algorithm), JarvisPatrick (Jarvis\[Dash]Patrick clustering algorithm), KMeans (k-means clustering), MeanShift (mean-shift clustering), KMedoids (k-medoids partitioning), SpanningTree (minimum spanning tree clustering), Spectral (spectral clustering), and GaussianMixture (Gaussian mixture model).

## Choosing a Distance Function

In addition to the method option, there is also the DistanceFunction, which was given the value of Automatic. This option defines how the distance between the points is calculated. In general, when you choose automatic, the square Euclidean distance is used ( $\sum (y_i - x_i)^2$ ). There are also other values for the distance function,

Euclidean distance ( $\sum \sqrt{(y_i - x_i)^2}$ ), Manhattan distance ( $\sum |x_i - y_i|$ ), Chessboard distance, or Chebyshev distance ( $(|x_i - y_i|)$ ), among others. Now that you know how the clusters are identified, you want to know the centroid of each one. For this it is necessary to

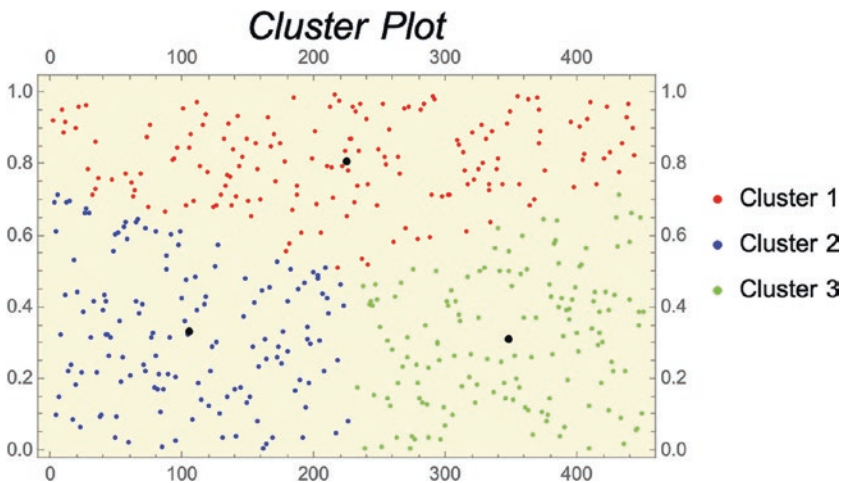


calculate the mean of the points of the clusters. The centroid of a series of points is obtained from the expression  $\left(\mu = \sum \frac{x_i}{n}\right)$ , which can be interpreted as the average of the points. For the calculation, you extract the data from each cluster and calculate its arithmetic mean.

```
In[79]:= {cluster1Centroid, cluster2Centroid, cluster3Centroid} = {N@Mean@
clusters[[1, All]], N@Mean@clusters[[2, All]], N@Mean@clusters[[3, All]]}
Out[79]= {{224.806, 0.810328}, {105.14, 0.331805}, {347.514, 0.31097}}
```

Let's plot the clusters with their centroids to visualize how the points are classified for each centroid (see Figure 8-31).

```
In[99]:= clusterPlot = ListPlot[clusters, PlotStyle -> {Red, Blue, Green},
PlotLegends -> {"Cluster 1", "Cluster 2", "Cluster 3"}];
centroidPlot = ListPlot[{cluster1Centroid, cluster2Centroid, cluster3Centroid},
PlotStyle -> Black];
Show[{clusterPlot, centroidPlot}, Prolog -> {LightYellow, Rectangle[Scaled[{0, 0},
Scaled[{1, 1}]]], Scaled[{1, 1}]}], Frame -> True, FrameTicks -> All, PlotLabel -> Style["Cluster
Plot", Italic, 20, Black]]
Out[100]=
```

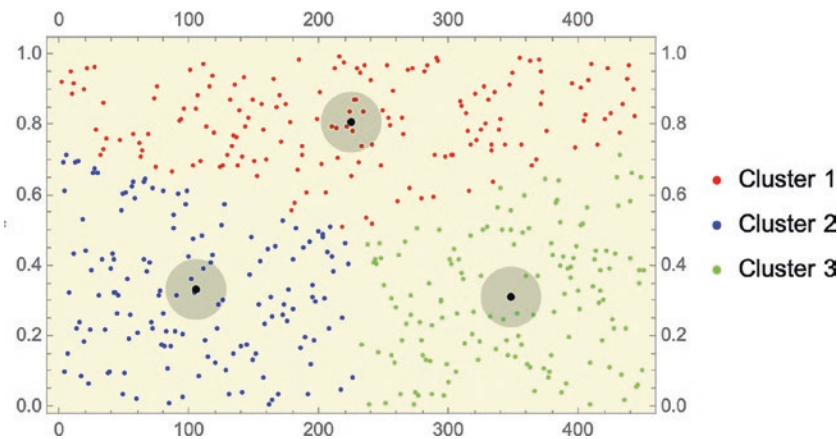


**Figure 8-31.** 2D scatter plot of the three clusters identified with their respective centroids



To make sure the first cluster corresponds to the red points, try using `ListPlot` to plot the points contained in `clusters[[1, All]]`, as well as those in the second cluster (blue) and third cluster (green). Alternatively, you can highlight the area of the centroids by adding the `Epilog` option to the plot. `Epilog` is another graphic option like `Prolog`, but you can use it to highlight the location of the centroid points (see Figure 8-32).

```
In[101]:= Show[{clusterPlot, centroidPlot}, Prolog -> {LightYellow,
Rectangle[Scaled[{0, 0}], Scaled[{1, 1}]]}, Frame -> True,
FrameTicks -> All, Epilog -> {Opacity[0.2], PointSize[0.1],
Point[cluster1Centroid], Point[cluster2Centroid],
Point[cluster3Centroid]}]
Out[101]=
```



**Figure 8-32.** 2D scatter plot of the three clusters identified with their respective centroids

## Identifying Classes

Once the clusters are identified by the command `FindClusters`, you can use the `ClusteringComponents` command to label or identify the different classes found. You must specify the number of clusters and where to look for the clusters within the `ClusteringComponents` command since there are several ways to use `ClusteringComponents`.

[illegible]

In this way, numbers that correspond to the three classes appear. The command only identifies three types of classes; it does not mention what each class means. This is because cluster methods are often performed on unlabeled data, so interpretation is part of the analysis. Let's count how many elements of each class you have.

```
In[103]:= Flatten[classes]//Counts
Out[103]= <|1->174,2->132,3->144|>
```

The command returns that class one contains 174, class two contains 132, and class three contains 144. One point to clarify is why the clusters identified with FindClusters and ClusteringComponents defer. This is because by setting the automatic option in the distance function, you are telling Mathematica to find the optimal distance function. Depending on the data, one function might gather elements in different forms, as you see later.

## K-Means Clustering

Thus far, you have seen how to search for clusters in a generic way. This section focuses on the k-means method. The k-means is a technique to find and classify data groups (k) so that the elements that share similar characteristics are grouped similarly for the

opposite case (not similar characteristics). The method calculates the distance between the data for a centroid to distinguish whether the data contain similarities. The elements that have less distance between them is those that share similarities. This technique is an iterative process in which the groups are adjusted until they reach a convergence. The k-means method, a simple algorithm, makes a classification employing specific partitions in different groups, where each point or observation belongs to the group. Clustering is done by minimizing the sum of the distances between each object and the centroid of its group. The k-means clustering technique tries to build the clusters to have the least variation within a group. This is done by minimizing the expression  $(C_i) = \sum_{x_j \in C_i} |x_j - \mu_i|^2$ , where  $C_i$  represents the  $i$ th cluster,  $x_j$  represents the points, and  $\mu_i$  represents the centroid of each cluster. The square term of the function is the distance function; the most used is the square Euclidean distance, as in this case.

To learn more about the mathematical foundation behind this technique, consult the reference *An Introduction to Statistical Learning: With Applications in R* by Gareth James, Daniela Witten, Trevor Hastie, and Robert Tibshirani. (1st ed. 2013, Corr. 7th printing 2017 ed.: Springer).

The Fisher's Irises dataset in `ExampleData` is used in the following example. Recalling the dataset's features, execute the following code.

```
In[104]:= ExampleData[{"Statistics","FisherIris"},"ColumnDescriptions"]
Out[104]= {Sepal length in cm.,Sepal width in cm.,Petal length in cm.,Petal
width in cm.,Species of iris}
```

Let's extract the dataset and assign the variable `iris` to it.

```
In[105]:= iris=ExampleData[{"Statistics","FisherIris"}];
Short[iris,6]
Out[106]//Short= {{5.1,3.5,1.4,0.2,setosa},{4.9,3.,1.4,0.2,setosa},{4.7,3.2,
1.3,0.2,setosa},{4.6,3.1,1.5,0.2,setosa},{5.,3.6,1.4,0.2,setosa},{5.4,3.9,
1.7,0.4,setosa},{4.6,3.4,1.4,0.3,setosa},<<136>>,{6.8,3.2,5.9,2.3,virginica},
{6.7,3.3,5.7,2.5,virginica},{6.7,3.,5.2,2.3,virginica},{6.3,2.5,5.,1.9,
virginica},{6.5,3.,5.2,2.,virginica},{6.2,3.4,5.4,2.3,virginica},{5.9,3.,
5.1,1.8,virginica}}
```

## Dimensionality Reduction

Since the iris dataset consists of four features classified into three species types, you use the PCA method, as this method is used to reduce high-dimensionality problems. In this case, you want to represent these features through two main components. For this, you proceed to standardize the data—that is, they have zero mean and one standard deviation since the variables with larger variance are more likely to affect the PCA.

```
In[107]:= sT=Standardize[iris[[All,{1,2,3,4}]]];(*Showing only the first
4 terms*)
```

```
%[[1;;4]]//TableForm
```

```
Out[108]//TableForm=  -0.897674    1.0156    -1.33575    -1.31105
-1.1392    -0.131539    -1.33575    -1.31105
-1.38073    0.327318    -1.3924    -1.31105
-1.50149    0.0978893    -1.2791    -1.31105
```

There are two ways to do the process, either using the `DimensionReduce` command or the `DimensionReduction` command, which are used to reduce the dimensions of the data. The difference between the two is that the first returns the values as a list. The second returns a `DimensionReducerFunction` (see Figure 8-33) as output, as in the case of `Predict` and `Classify`. Both belong to the Wolfram Language special functions for machine learning. For this case, you use the `DimensionReduction` command. Since you have the data, you introduce the standardized data as arguments, followed by specified target dimensions (2), with the `PrincipalComponentAnalysis` method. This gives you the `DimensionReducerFunction` that assigns the name DR.

```
In[109]:= dR=DimensionReduction[sT,2,Method->"PrincipalComponentsAnalysis"]
Out[109]=
```

`DimensionReducerFunction` [  Input type: NumericalVector (4)  
Output dimension: 2  
Method: PrincipalComponentsAnalysis  
Number of training examples: 150 ]

**Figure 8-33.** *DimensionReductionFunction* object

The properties of the function are “ReducedVectors” (list of reduced vectors), “OriginalData” (deduction from the original data list given the reduced vectors), “ReconstructedData” (data reconstruction by reduction and inversion), “ImputedData” (missing values replaced by imputed ones). You call the standardized data values function, showing the first five. The coordinates x and y are for the principal components 1 and 2, respectively.

```
In[110]:= pCA=dR[sT,"ReducedVectors"]; TableForm[%[[1;;5]],TableHeadings
->{None, {"First principal component","Second Principal component"}},
TableAlignments->Center]
Out[111]//TableForm= First principal component    Second Principal component
2.2647      -0.480027
2.08096     0.674134
2.36423     0.341908
2.29938     0.597395
2.38984     -0.646835
```

This calculates the variance of each component, followed by the total to find the proportion of variance explained. PC1 represents 76% of the data dispersion, and PC2 represents 23%. To obtain the accumulated percentage, you add the variations of each component. To view more depth about the proportion of variation, refer to *An Introduction to Statistical Learning: With Applications in R* by G. James, D. Witten, T. Hastie, and R. Tibshirani (Springer, 2017).

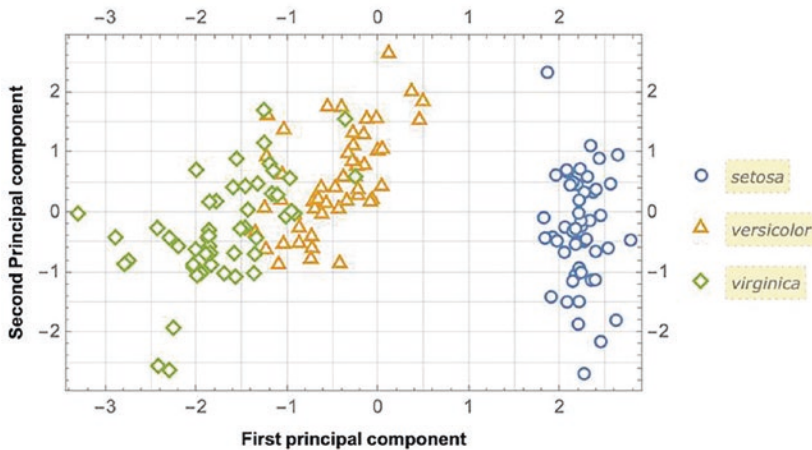
```
In[112]:= Variance@pCA[[All, All]]/Total@Variance@pCA[[All, All]]
// TableForm[#, TableHeadings -> {"First PC variation", "Second PC
variation"}, None]] &
Out[112]//TableForm=
First PC variation    | 0.761507
Second PC variation   | 0.238493
```

You look at the plot (see Figure 8-34) of the main components made by the previous process. If you look over the complete iris data from the ExampleData, the first 50 elements correspond to the setosa species, the next 50 to versicolor, and the last 50 to virginica.

```

In[113]:= labels={Style["First principal component", Black, Bold],
Style["Second Principal component",Black,Bold]};ListPlot[{pCA[[1 ;; 50]],
pCA[[51 ;; 100]], pCA[[100 ;; 150]]},PlotLegends->Placed
[{Placeholder["setosa"], Placeholder["versicolor"], Placeholder
["virginica"]}, Right], PlotMarkers -> "OpenMarkers", GridLines -> All,
Frame -> True, Axes -> False, FrameTicks -> All, FrameLabel -> labels]
Out[114]=

```



**Figure 8-34.** Scatter plot of the two principal components

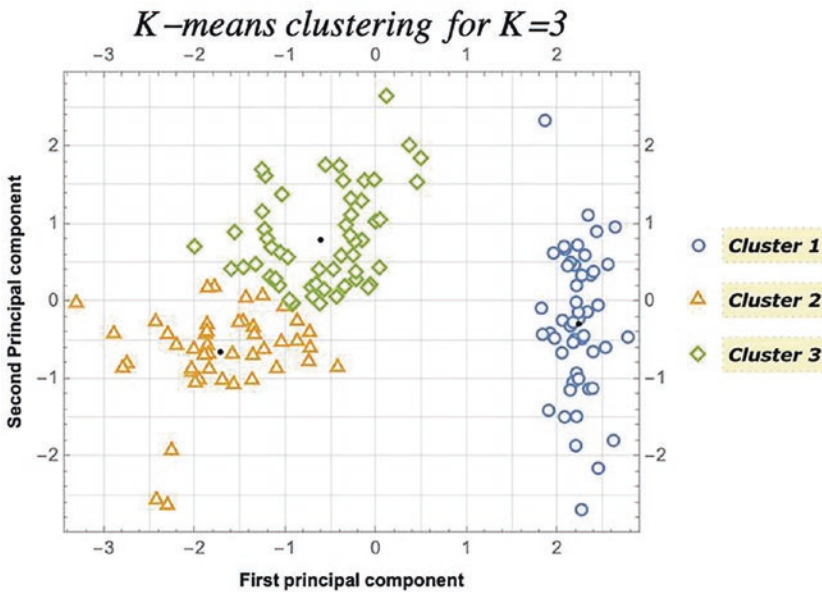
## Applying K-Means

Now, let's find the clusters with k-means using the Manhattan distance. You assume that the data can be divided into three clusters by specifying to look for three clusters. You know the original data belongs to three species (setosa, versicolor, and virginica). The plot of the clusters is shown here (see Figure 8-35), with their respective centroids. When choosing the k-means method, suboptions can be added, like InitialCentroids. Costum start centroids (a list of centroid coordinates) can be typed, or you can leave the automatic option. To enter the centroids coordinates, you use the following form Method  $\rightarrow$  {"KMeans," InitialCentroids"  $\rightarrow$  {{x1, y1}, {x2, y2}, {x3, y3} ... }}, where x1, y1 represent the centroid of the C1 (cluster 1). Initial centroids are not given to the command FindClusters to keep some randomness.

```

In[115]:= clstr = FindClusters[pCA, 3, Method ->
"KMeans", DistanceFunction -> SquaredEuclideanDistance, RandomSeeding
-> 8888];ListPlot[clstr, PlotRange -> All, Frame -> True, AspectRatio ->
0.8, Axes -> False, PlotStyle -> {ColorData[97, 1], ColorData[97, 2],
ColorData[97, 3]}, PlotLabel -> Style["K-
means clustering for K=3", FontFamily -> "Times", Black, 20, Italic],
FrameTicks -> All, PlotLegends -> Placed[{Placeholder[Style["Cluster 1",
Bold, Black, 10]], Placeholder[Style["Cluster 2", Bold, Black, 10]],
Placeholder[Style["Cluster 3", Bold, Black, 10]]}, Right], PlotMarkers
-> "OpenMarkers", FrameLabel -> labels, GridLines -> All, Epilog ->
{Opacity[1], PointSize[0.01], Point[Mean@clstr[[1, All]]], Point[Mean@
clstr[[2, All]]], Point[Mean@clstr[[3, All]]]}]
Out[115]=

```



**Figure 8-35.** 3 clusters identified of the two principal components

In Figure 8-35, the method identifies the left points as a single cluster (setosa specie), whereas some points between clusters 2 and 3 might be misclassified.

## Changing the Distance Function

Changing the `DistanceFunction` can modify how the clusters are arranged; the following code shows the plot for  $k = 3$  and choosing a different distance function. In the next block of code, the computation of the clusters is made for the same  $k$  (3), with a different distance function, and stored into their respective variables. Then, the clusters are plotted (see Figure 8-36) for each of the different distance functions, and finally, they are displayed within a graphic grid.

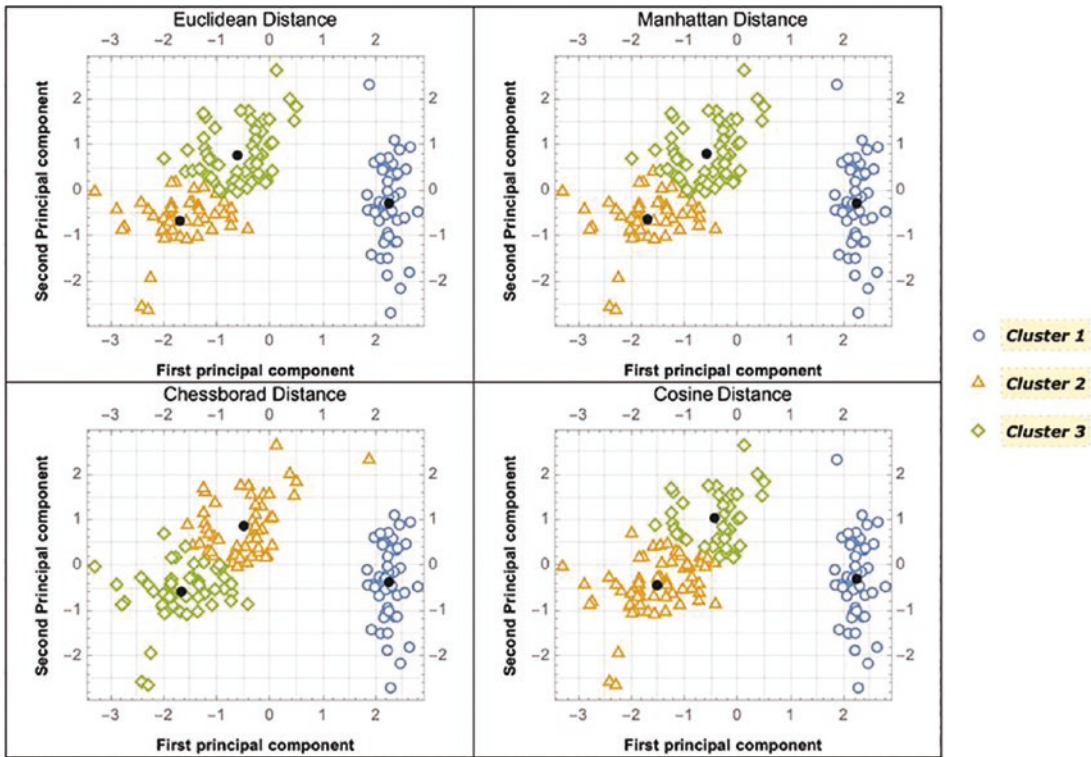
```
In[116]:= clusteringPlot[distanceName_, distanceFunction_]
:= Module[{clusters, pltTitles, points}, clusters = FindClusters[pCA,
3, PerformanceGoal -> "Quality", Method -> "KMeans", DistanceFunction
-> distanceFunction, RandomSeeding -> 8888]; points = Point[Mean[#]]
& /@ clusters; pltTitles = distanceName; ListPlot[clusters, Frame ->
True, AspectRatio -> 0.8, PlotMarkers -> "OpenMarkers", PlotStyle ->
{ColorData[97, 1], ColorData[97, 2], ColorData[97, 3]}, GridLines ->
All, PlotRange -> Automatic, ImageSize -> 300, FrameLabel -> labels,
Axes -> False, FrameTicks -> All, Epilog -> {Opacity@1, PointSize@0.03,
points}, PlotLabel -> Style[pltTitles, Black]]]
```

```
eDplt = clusteringPlot["Euclidean Distance", EuclideanDistance];
mhDplt = clusteringPlot["Manhattan Distance", ManhattanDistance];
chDplt = clusteringPlot["Chessboard Distance", ChessboardDistance];
cosDplt = clusteringPlot["Cosine Distance", CosineDistance];
```

```
legendsText = {Placeholder[Style["Cluster 1", Bold,
Black, 10]], Placeholder[Style["Cluster 2", Bold,
Black, 10]], Placeholder[Style["Cluster 3", Bold, Black,
10]]}; Labeled[Legended[GraphicsGrid[{{eDplt, mhDplt},
{chDplt, cosDplt}}, Frame -> All, Background -> White, Spacings -> 1],
PointLegend[{ColorData[97, 1], ColorData[97, 2], ColorData[97, 3]},
legendsText, LegendMarkers -> "OpenMarkers"]], Style["K-means clustering
for K=3", FontFamily -> "Times", Black, 20, Italic], Top]
Out[117]=
```



### *K-means clustering for $K=3$*



**Figure 8-36.** *K-means clustering for  $K = 3$ , for different distance functions*

The clusters can have different arrangements with different distance functions; one thing to note also is that the cluster's centroids change in each of the subfigures.

## Different k's

Having seen that for different distance functions, the clusters can vary, let's now construct the process but with different k's—that is, for  $k = 2, 3, 4$ , and  $5$ , as exhibited in Figure 8-37.

```
In[117]:= findKClusters[k_, PCA_] := FindClusters[PCA, k,
PerformanceGoal -> "Speed", Method -> "KMeans", DistanceFunction ->
SquaredEuclideanDistance, RandomSeeding -> 8888];

plotKClusters[k_, clusters_] := ListPlot[clusters, Frame -> True,
```

```

AspectRatio -> 0.8, PlotMarkers -> "OpenMarkers", PlotStyle ->
ColorData[97, "ColorList"][[;; k]], GridLines -> All, PlotRange ->
Automatic, ImageSize -> 260, FrameLabel -> labels, Axes -> False,
FrameTicks -> All, Epilog -> {Opacity@1, PointSize@0.015, Point[Mean
/@ clusters]}}, PlotLabel -> Style["K=" <> ToString[k], Black]];

kValues = {2, 3, 4, 5};
kClusters = findKClusters[#, pCA] & /@ kValues;
kPlots = plotKClusters[#, kClusters[[#2]]] & @@@ Transpose[{kValues, Range@
Length@kValues}]];

legendsText2 = {Placeholder[Style["Cluster " <> ToString[#], Bold, Black,
10]]} & /@ Range@5;
Labeled[Labeled[GraphicsGrid[Partition[kPlots, 2], Frame ->
All, Background -> White, Spacings -> 1], PointLegend[ColorData[97,
"ColorList"][[;; 5]], legendsText2, LegendMarkers ->
"OpenMarkers"]], Style["K-means clustering for K=2,3,4,5", FontFamily ->
"Times", Black, 20, Italic], Top]
Out[120]=

```



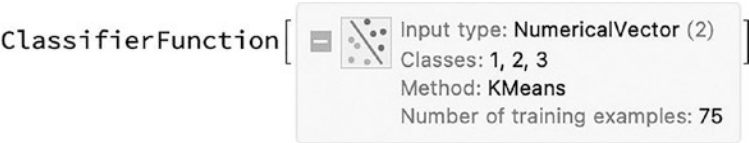
Given a clustering problem, the k-means technique is meant to be used for unlabeled data—that is, data without defined categories. Some factors that can alter the operation of the method include the following.

- The spread, or how far apart the points are. This is reflected if the data contains outliers or are in various scales, which can be erroneously classified as part of a cluster when the opposite is observed visually.
- The dimensionality of the data. Given that more information and features are often added to the model, the number of dimensions grows, leading to the “curse of dimensionality.” This type of problem can be solved using data transformation methods, as in the example seen from PCA, but with some restrictions since the PCA method can lose sensitive information on the features.
- The value of k is determined manually, but when there are high-cost function values, it can be interpreted that the intra-cluster variation is high. With low-cost function values, the intra-cluster variation is low. The last two assumptions can also be attributed to the fact that for lower values of k, many observations can be grouped into large individual clusters. For high values of k observations, they can be a proper group.

## Cluster Classify

Another command that belongs to the cluster functions is called `ClusterClassify` (see Figure 8-38). This command works in the same way as `Classify` does. The following example uses this command to see how the k-means cluster classifies the species based on Sepal length and Sepal width. Split the data into halves when you randomly sample.

```
In[122]:= BlockRandom[
SeedRandom[88888];
RandomSample[iris[[All,{1,2}]]];]
trainingSet=%[[1;;75]];
testSet=%[[76;;150]];
In[123]:= cC=ClusterClassify[trainigSet,3,Method->"KMeans",
DistanceFunction->Automatic,PerformanceGoal->"Speed",RandomSeeding->8888 ]
Out[123]=
```



**Figure 8-38.** *ClassifierFunction of the cluster classification model*

Figure 8-38 shows the details of the cluster classification model. The input vector is a numerical vector, the number of classes (three), the method, and the number of training examples.

**Note** To correctly use the k-means method, the number of clusters needs to be specified; otherwise, the command does not execute correctly.

Use the Information command to see the classifier information (see Figure 8-39).

```
In[124]:= Information[cC]  
Out[124]=
```

Classifier information	
Data type	NumericalVector (2)
Classes	1, 2, 3
Method	KMeans
Single evaluation time	1.79 ms/example
Batch evaluation speed	228. examples/ms
Model memory	73.9 kB
Training examples used	75 examples
Training time	30.4 ms

**Figure 8-39.** *Classifier information for k-means*

More detailed information about the classifier function is shown in Figure 8-39. To get the complete list of properties, type “Properties” as a second argument. Many metrics, such as BatchEvaluationSpeed, BatchEvaluationTime, and TrainingTime, can compare times with different methods.

```
In[125]:= Information[cC,"Properties"]
Out[125]={AcceptanceThreshold,AnomalyDetector,BatchEvaluationSpeed,BatchEvaluationTime,Calibrated,Classes,ClassNumber,ClassPriors,DistanceFunction,EvaluationTime,ExampleNumber,FeatureExtractor,FeatureNames,FeatureNumber,FeatureTypes,FunctionMemory,FunctionProperties,IndeterminateThreshold,LearningCurve,MaxTrainingMemory,Method,MethodDescription,MethodOption,MethodParameters,MissingSynthesizer,PerformanceGoal,Properties,TrainingClassPriors,TrainingTime,UtilityFunction}
```

Let’s now get the information about the classes identified from the cluster classifier, the number of classes, distance function, feature names, and the training class probabilities.

```
In[126]:=Information[cC,#]&/{ "Classes","ClassNumber","DistanceFunction",
"FeatureNames","TrainingClassPriors"}
Out[126]= {{1,2,3},3,EuclideanDistance,{f1},<|1->0.333333,2->0.293333,
3->0.373333|>}
```

There are three classes: class 1, class 2, and class 3. The distance function used is EuclideanDistance, and the name f1 refers to the numeric vector features. A simple example is chosen by choosing a sepal length of 1 and a sepal width of 2 to show the different properties that can be used when testing the data, shown in the dataset form (see Figure 8-40). The example is first written, followed by the properties Decision (cluster that belongs to the example), Distribution (categorical distribution object for histogram plots), ExpectedUtilities (expected probabilities and indeterminate threshold), LogProbabilities (log probabilities), Probabilities (probabilities of the test data based on classes), and TopProbabilities (best probabilities for the test data).

```
In[127]:=Dataset[AssociationMap[cC[{1,2},#]&,{ "Decision","Distribution",
"ExpectedUtilities","LogProbabilities","Probabilities","TopProbabilities"}]]
Out[127]=
```

Decision	3
Distribution	CategoricalDistribution[ <div>Input type: Scalar Categories: 1 2 3</div> ]
ExpectedUtilities	$\langle   1 \rightarrow 0.0238517, 2 \rightarrow 2.72758 \times 10^{-16}, 3 \rightarrow 0.976148, \text{Indeterminate} \rightarrow 0.   \rangle$
LogProbabilities	$\langle   1 \rightarrow -3.7359, 2 \rightarrow -35.8379, 3 \rightarrow -0.0241407   \rangle$
Probabilities	$\langle   1 \rightarrow 0.0238517, 2 \rightarrow 2.72758 \times 10^{-16}, 3 \rightarrow 0.976148   \rangle$
TopProbabilities	{3 $\rightarrow$ 0.976148}

Figure 8-40. The dataset of the simple Iris example

The example belongs to the third cluster and that the associated probability is  $3 \rightarrow 0.976148$ . Look at the rest of the data and plot the cluster classification. The classified data plot is shown in Figure 8-41.

```
In[128]:= ListPlot[Pick[testSet,cC[testSet],#]&/@{1,2,3},
PlotMarkers->"OpenMarkers",GridLines->Automatic,PlotLegends->
{Placeholder[Style["Cluster 1",Bold,Black,10]],Placeholder[Style["Cluster
2",Bold,Black,10]],Placeholder[Style["Cluster 3",Bold,Black,10]]},
Frame->True,FrameTicks->All,FrameLabel->{"Sepal Length","Sepal Width"}]
Out[128]=
```

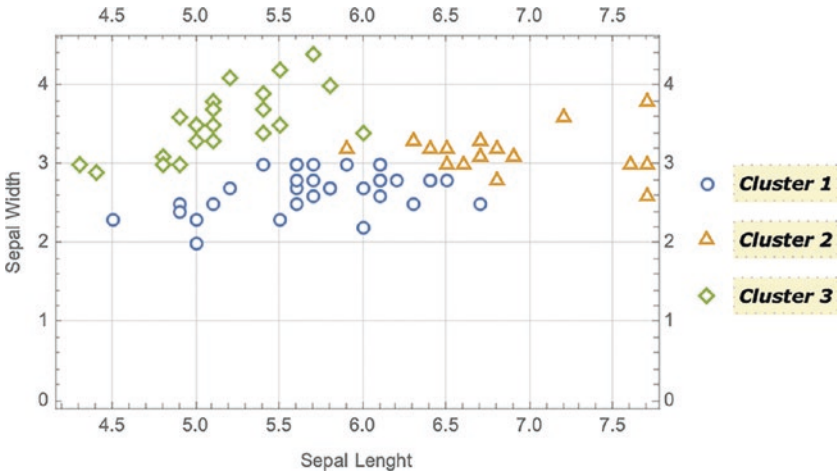
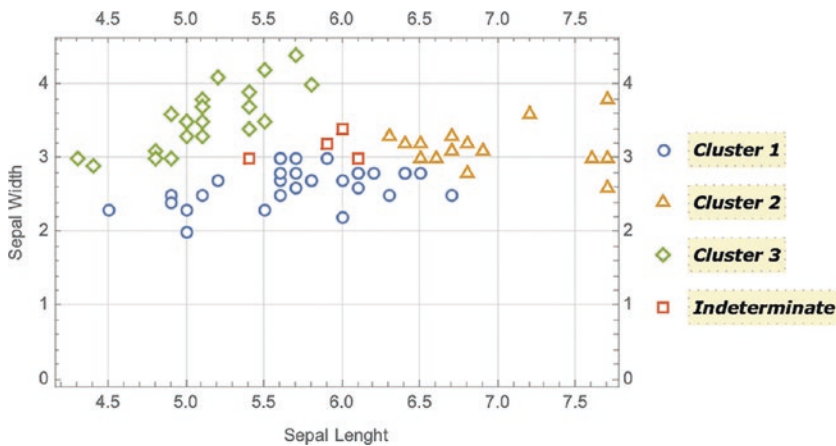


Figure 8-41. Cluster classification on the example of the iris data for the first two features



As a complement, a probability restriction for values below an established probability value can be added with `IndeterminateThreshold`, as depicted in Figure 8-42.

```
In[129]:= ListPlot[Pick[testSet,CC[testSet,IndeterminateThreshold->
0.6],#]&@{1,2,3,Indeterminate},PlotMarkers->"OpenMarkers",PlotLegends->
{Placeholder[Style["Cluster 1",Bold,Black,10]],Placeholder[Style["Cluster
2",Bold,Black,10]],Placeholder[Style["Cluster 3",Bold,Black,10]],Placeholder
[Style["Indeterminate",Bold,Black,10]]},Frame->True,FrameTicks->
All,FrameLabel->{"Sepal Length","Sepal Width"},GridLines->Automatic]
Out[129]=
```



**Figure 8-42.** Cluster classification on the example of the iris data for the first two features with a probability restriction

## Summary

The first part of the chapter discussed machine learning, the gradient descent algorithm, and its comprehensive implementation. Then, the linear regression model was introduced by exploring the Boston dataset and the guide to creating, measuring, and refining the created model. This previous process is also carried out for the logistic regression but with the Titanic dataset. As the chapter concluded, you learned about data clustering and k-means clustering.



## CHAPTER 9

# Neural Networks with the Wolfram Language

This chapter starts with the basic foundations of the neural network framework in the Wolfram Language. The chapter begins with the concepts of layers, how to use the commands for different layers, and the most common layers. You learn how to enter data into the layers by the net port and the different forms of equivalent expression of the layers. This topic is followed by how to distinguish different layers by their symbol. You see that layers can have multiple options that enable them to have various specifications by viewing the concept of a layer in the Wolfram Language scheme, comparing different layers with different purposes, and performing different computations. You also achieve this by looking at the various activation functions supported by the Wolfram Language and inspecting the plots of each function in addition to different syntax forms. Next, you learn about encoders and decoders and how these tools are used to construct a neural network model, depending on the task to be fulfilled. You then learn how these encoders and decoders are used to convert different data types to numeric arrays and how to convert the numeric arrays back to the initial data. You introduce the concept of a container, what it means for the created models, and what types exist. You see how to handle and build containers with different commands and graphically visualize the created model. You see how the Wolfram Neural Net Framework supports MXNet-related operations and how to export a network to the format of the MXNet operation.

# Layers

It is necessary to understand that neural networks, in general and in the Wolfram Language, are built from layers. A layer is a term that can be applied to a collection of nodes that operate together at a specific level within the neural network. The layer is an essential and straightforward member for constructing a neural network.

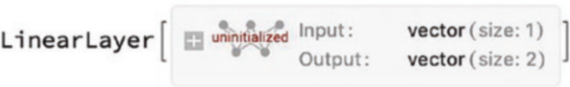
## Input Data

The data handled by the layers is of a numeric type and not of another kind. Input variables can be vectors, a unidimensional list, matrixes, a two-dimensional list, arrays, a list of lists, or any other numeric tensor. These input variables can be either features or attributes of the dataset of study, with a known or multidimensional shape. These types of input attributes are associated with the input layer, for which the feature size, in turn, must be equal to the input size of a layer, but not every layer receives the same input and returns the same output; every input varies depending on the type of layer to be used. This definition is one of the most basic ideas in neural networks since they are a crucial component of the whole structure that involves the term neural network. A remark here is to distinguish input from input layer since they do not mean the same.

## Linear Layer

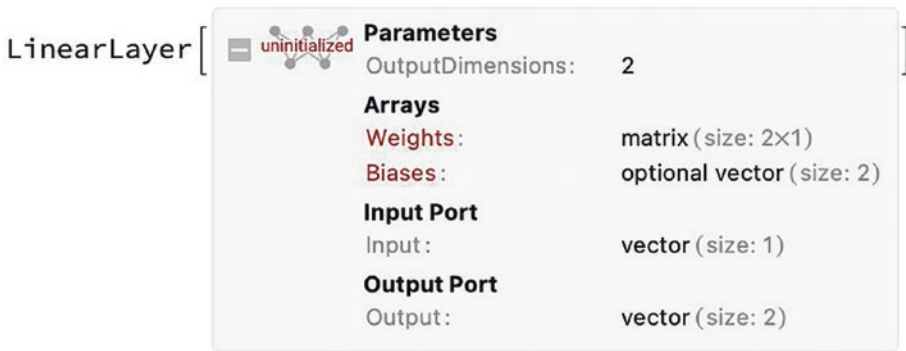
A linear layer is the most common and widely used layer in a neural network. To build the simplest layer in the Wolfram Language, use the `LinearLayer` command.

```
In[1]:= LinearLayer["Input"->1,"Output"->2]
Out[1]=
```



**Figure 9-1.** *LinearLayer* object

Figure 9-1 represents the `LinearLayer` object in the Wolfram Language. Clicking the plus icon shows the internal parameters, including details about the layer port’s input and output and array rank of the weights and biases of the linear layer, as shown in Figure 9-2.



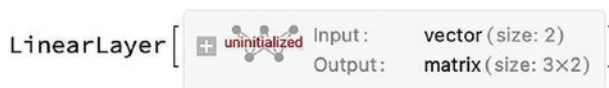
**Figure 9-2.** Expanded *LinearLayer* object

Each layer has an input port and an output port. Each port has an associated size of what is entering the layer and what is going out. In the latter case, a vector of size one is entering, and the layer returns a vector of size two.

## Weights and Biases

The general form of a linear layer is given by the following expression of the dot product  $\mathbf{w} \cdot \mathbf{x} + \mathbf{b}$ , where  $\mathbf{x}$  is the data vector,  $\mathbf{w}$  represents the matrix of the weights, and  $\mathbf{b}$  is the vector of the biases. Linear layers have other associated names, like fully connected layers, as in the MXnet framework. The input of the layers in the Wolfram Language receives numerical tensors as input—that is, they only act on numerical arrays. To explicitly enter the size of input and output, you write the form of the input port and the output port followed by different options: “Input” or “Output”  $\rightarrow$  {size, Options.}. Options include defining a real number (Real), a vector of form  $n$  (single number  $n$ ), an array ( $\{n_1 * n_2 * n_3 \dots\}$ ), or a NetEncoder, which you see later. Following are some equivalent ways to write layers, as depicted in Figure 9-3.

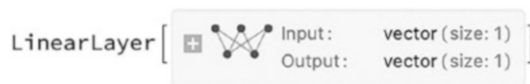
```
In[2]:= LinearLayer["Input" -> {2, "Real"}, "Output" -> {3, 1}]
Out[2]=
```



**Figure 9-3.** *LinearLayer* with different input and output rank arrays

As shown in Figure 9-3, the layer receives a vector of size two (list of length 2), comprised of real numbers, and the output is a matrix of the shape  $3 \times 2$ . When a real number is specified within the Wolfram Neural Network Framework, it works with the precision of a Real32. When no arguments are added to the layer, the input and output shapes are inferred. To manually assign the weights and biases, write “Weights” → number, “Biases” → number; None is also available for no weights or biases. This is shown in the following example, where weights and biases are set to a fixed value of 1 and 2 (see Figure 9-4).

```
In[3]:= LinearLayer["Input"-> 1,"Output"-> 1,"Weights"-> 1,"Biases"-> 2]
Out[3]=
```

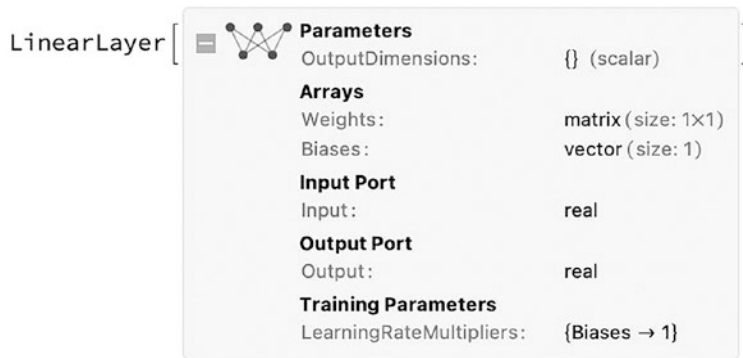


**Figure 9-4.** *Initialized linear layer, with fixed biases and weights*

## Initializing a Layer

Another command allows you to initialize the layer with random values: NetInitialize. So, to establish hold values of weights or biases, you can also use the LearningRateMultipliers option (see Figure 9-5). Besides this, LearningRateMultipliers also mark the rate at which a layer learns during the training phase.

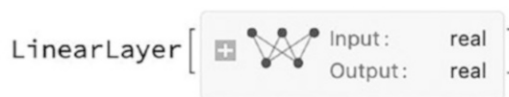
```
In[4]:= NetInitialize[LinearLayer["Input"-> "Real","Output"->
"Real",LearningRateMultipliers->{"Biases"->1}]]
Out[4]=
```



**Figure 9-5.** *LinearLayer with training parameters*

When a layer is initialized, the uninitialized text disappears. If you observe the properties of the new layer, they appear within the training parameters where fixed biases have been established, and a learning rate has been set. The options for `NetInitialize` are `Method` and `RandomSeeding`. The available methods are `Kaiming`, `Xavier`, `Orthogonal` (orthogonal weights), and `Random` (weights selection from a distribution). For example, you can use the `Xavier` initialization sampling from a normal distribution, as seen in Figure 9-6.

```
In[5]:= NetInitialize[LinearLayer["Input"-> "Real", "Output"->
"Real", LearningRateMultipliers->{"Biases"->1}], Method->
{"Xavier", "Distribution"->"Normal"}, RandomSeeding->888]
Out[5]=
```



**Figure 9-6.** *LinearLayer initialized with the Xavier method*

---

**Note** The `Option` command is recommended to see the options set for a layer.

---

Despite being able to establish the weights and biases manually, it is advisable to start the layer with random values to maintain a certain level of complexity in the overall structure of a model since, on the contrary, this could have an impact on the creation of a neural network that does not make accurate predictions for non-linear behavior.

## Retrieving Data

NetExtract retrieves the value of the weights and biases in the form NetExtract [net, {level1, level2, ...}]. The weights and bias parameters of the linear layers are packed in NumericArray objects (see Figure 9-7). This object has the values, dimensions, and type of the values in the layer. NetExtract also serves to extract layers of a network with many layers. NumericArrays are used in the Wolfram Language to reduce memory consumption and computation time.

```
In[6]:= linearL=NetInitialize[LinearLayer[2, "Input"->
1],RandomSeeding->888];
NetExtract[linearL,#]&/@{"Weights","Biases"}//TableForm
Out[7]//TableForm=
```



**Figure 9-7.** *Weights and biases of a linear layer*

With Normal, you convert them to lists.

```
In[8]:=TableForm[SetPrecision[{{Normal[NetExtract[linearL,"Weights"]]},
{Normal[NetExtract[linearL,"Biases"]]}},3],TableHeadings->{{"Weights
->","Biases ->"},None}]
Out[8]//TableForm=
Weights ->    -0.779
0.0435
Biases  ->    0
0
```

For instance, a layer can receive a length of one vector to produce an output vector of size 2.

```
In[9]:= linearL[4]
Out[9]= {-3.11505,0.174007}
```

The layer can only be evaluated when input is introduced in the appropriate shape.

```
In[10]:= linearL[{88,99}]
```

During evaluation of In[10]:= LinearLayer::invindata1: Data supplied to port "Input" was a length-2 vector of real numbers, but expected a length-1 vector.

```
Out[10]= $Failed
```

The weights and biases are the parameters that the model must learn from, which can be adapted based on the input data that the model receives, which is why it is initialized randomly since if you try to extract these values without initializing, you cannot because they have not been defined.

Layers have the property of being differentiable. It is achieved with NetPortGradient, which can represent the gradient of a net output for a port or a parameter. For example, give the derivative of the output concerning the input for a particular input value.

```
In[11]:= linearL[2,NetPortGradient["Input"]]
```

```
Out[11]= {-0.735261}
```

## Mean Squared Layer

Until now, you have seen the linear layer, which has various properties. Layers with the icon of a connected rhombus (see Figure 9-8), by contrast, do not contain any learnable parameters, like MeanSquaredLossLayer, AppendLayer, SummationLayer, DotLayer, ContrastiveLossLayer, and SoftmaxLayer, among others.

```
In[12]:= MeanSquaredLossLayer[]
```

```
Out[12]=
```



**Figure 9-8.** *MeanSquaredLossLayer*

`MeanSquaredLossLayer[]` has more than one input because this layer computes the mean squared loss, which is the following expression  $(1/n) \sum (\text{Input} - \text{Target})^2$ , and has the property that compares two numeric arrays. With the `MeanSquaredLossLayer`, the input/output ports' dimensions are entered in the same form as a linear layer, and the input and target values are entered as Associations.

```
In[13]:= MeanSquaredLossLayer["Input" -> {3, 2}, "Target" -> {3, 2}][
Association["Input" -> {{1, 2}, {2, 1}, {3, 2}}, "Target" -> {{2, 2},
{1, 1}, {1, 3}}]]
Out[13]= 1.16667
```

The latter example computes a `MeanSquaredLossLayer` for input/output dimensions of three rows and two columns or by defining first the layer and then applying the layer to the data.

---

**Note** Use the `Matrixform[{{1, 2}, {2, 1}, {3, 2}}]` command to verify the matrix shape of the data.

---

```
In[14]:= lossLayer=MeanSquaredLossLayer["Input" -> {3, 2}, "Target" -> {3, 2} ];
lossLayer@<|"Input" -> {{1, 2}, {2, 1}, {3, 2}}, "Target" -> {{2, 2}, {1, 1}, {1, 3}}|>
Out[15]= 1.16667
```

To get more details about a layer (see Figure 9-9), use the `Information` command.

```
In[16]:= Information[lossLayer]
Out[16]=
```



Net Information	
Layers Count	1
Arrays Count	0
Shared Arrays Count	0
Input Port Names	{Input, Target}
Output Port Names	{Loss}
Arrays Total Element Count	0
Arrays Total Size	0 B

**Figure 9-9.** Information about the loss layer To know the layer options, use the following

To know the layer options, use the following.

```
In[17]:= MeanSquaredLossLayer["Input"->"Real", "Target"->"Real"]//Options
Out[17]= {BatchSize->Automatic, NetEvaluationMode->Test, RandomSeeding->
Automatic, TargetDevice->CPU, WorkingPrecision->Real32}
```

The input port and target port options are similar to that of the linear layer with the different forms, Input  $\rightarrow$  Real,  $n$  (a form of a vector  $n$ ),  $\{n_1 \times n_2 \times n_3\}$  ... (an array of  $n$  dimensions), Varying (a vector or varying form) or a NetEncoder, but with the exception that the input and target must have the exact dimensions. A few forms of layers are shown in Figure 9-10.

```
In[18]:= {MeanSquaredLossLayer["Input"->"Varying", "Target"->"Varying"],
MeanSquaredLossLayer["Input"-> NetEncoder["Image"], "Target"-> NetEncoder["I
mage"]], MeanSquaredLossLayer["Input"->1, "Target"->1]}//Dataset
Out[18]=
```

MeanSquaredLossLayer [	<div><b>Input Ports</b> Input: vector of <math>n</math> scalars Target: vector of <math>n</math> scalars <b>Output Port</b> Loss: real</div>	]
MeanSquaredLossLayer [	<div><b>Input Ports</b> Input: image Target: image <b>Output Port</b> Loss: real</div>	]
MeanSquaredLossLayer [	<div><b>Input Ports</b> Input: vector (size: 1) Target: vector (size: 1) <b>Output Port</b> Loss: real</div>	]

**Figure 9-10.** Loss layers with different input and target forms

## Activation Functions

Activation functions are a crucial part of the construction of a neural network. The role of an activation function is to return an output from an established range, given an input. In the Wolfram Language, activation functions are treated as layers. The layer that is frequently used for activation function definition in the Wolfram Language neural net framework is the `ElementwiseLayer`. With this layer, you can represent layers that can apply a unary function to the input data elements—in other words, a function that receives only one argument. These functions are also known as activation functions. For example, one of the most common functions used is the hyperbolic tangent (`Tanh[x]`), shown in Figure 9-11.

```
In[19]:= ElementwiseLayer[Tanh[#]&](* Alternate form  
ElementwiseLayer[Tanh]*)  
Out[19]=
```



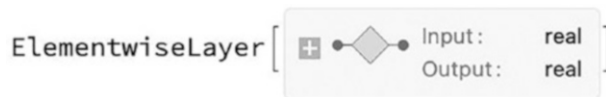
**Figure 9-11.** *Tanh[x] function layer*

Elementwise layers do not have learnable parameters. The pure function is used because layers cannot receive symbols. If the plus icon is clicked, detailed information about the ports and the parameters with the associated function, Tanh, are shown. Having defined an ElementwiseLayer, it can receive values like the other layers.

```
In[20]:= ElementwiseLayer[Tanh[#]&];
Table[%i],{i,-5,5}]
Out[21]= {-0.999909,-0.999329,-0.995055,-0.964028,-0.761594,0.,0.761594,
0.964028,0.995055,0.999329,0.999909}
```

When no input or output shape is given, the layer infers the type of data it receives or returns. For instance, by specifying only the input as real, Mathematica infer that the output is real (see Figure 9-12).

```
In[22]:= tanhLayer=ElementwiseLayer[Tanh,"Input"-> "Real"]
Out[22]=
```



**Figure 9-12.** *ElementwiseLayer with the same output as the input*

Or, this can be inferred by entering only the output (see Figure 9-13) for a rectified linear unit (ReLU).

```
In[23]:= rampLayer=ElementwiseLayer[Ramp,"Output"-> {1}]( *or ElementwiseLayer["ReLU","Output" -> "Varying"]*)
Out[23]=
```



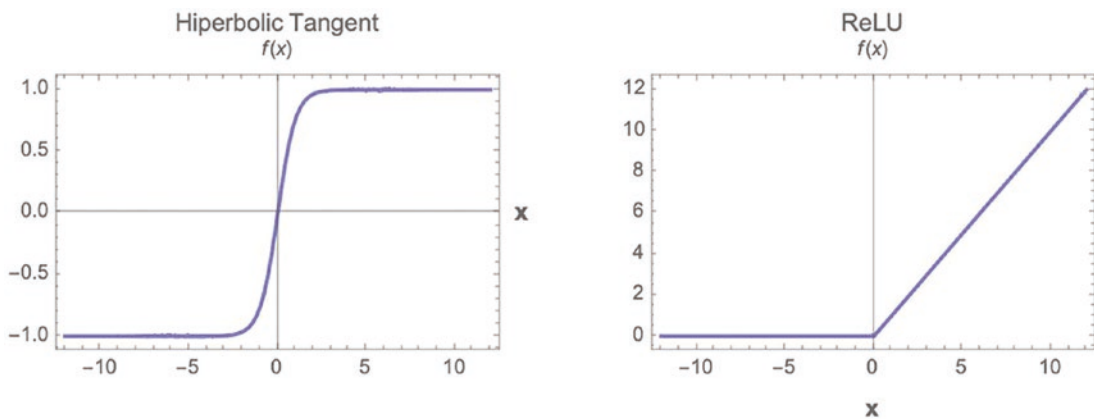
**Figure 9-13.** *Ramp function or ReLU*

**Note** Clicking the plus icon shows the elementwise layer's established function and the layer ports' details.

Every layer in the Wolfram Language can be run through a graphics processor unit (GPU) or a central processing unit (CPU) by specifying the `TargetDevice` option. It is essential to ensure your computer supports the specified functionality, so if you do not have a GPU, the compulsory target device is the CPU. For example, plot the previously created layers with the `TargetDevice` on the CPU (see Figure 9-14).

```
In[24]:= GraphicsRow@{Plot[tanhLayer[x, TargetDevice -> "CPU"], {x, -12, 12}, PlotLabel -> "Hiperbolic Tangent", AxesLabel -> {Style["x", Bold, 12], Style["f(x)", Italic]}, PlotStyle -> ColorData[97, 25], Frame -> True], Plot[rampLayer[x, TargetDevice -> "CPU"], {x, -12, 12}, PlotLabel -> "ReLU", AxesLabel -> {None, Style["f(x)", Italic]}, FrameLabel -> {{None, None}, {Style["x", Bold, 12], None}}, PlotStyle -> ColorData[97, 25], Frame -> True]}
```

Out[24]=



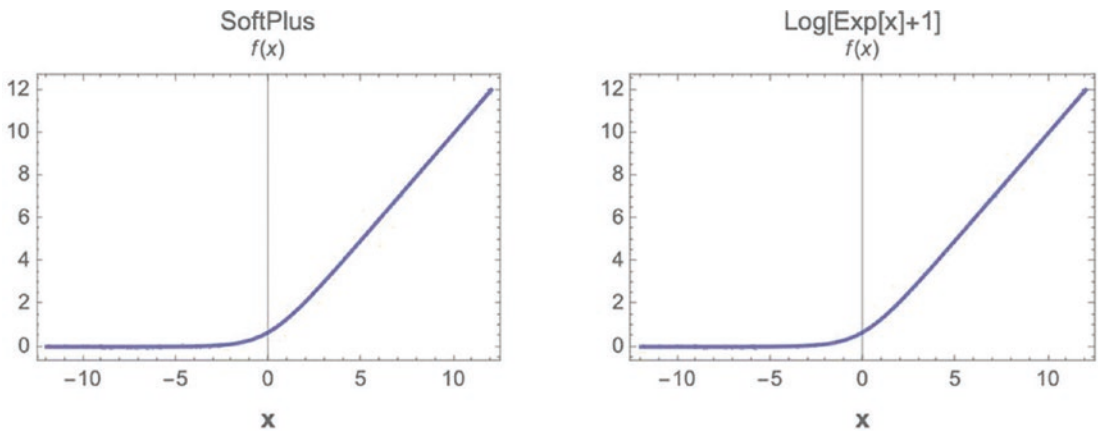
**Figure 9-14.**  $\tanh[x]$  and  $\text{Ramp}[x]$  activation functions

Other functions can be used by their name or Wolfram Language syntax—for instance, the `SoftPlus` function, as demonstrated in Figure 9-15.

```

In[25]:= GraphicsRow@{Plot[ElementwiseLayer["SoftPlus"][x, TargetDevice
-> "CPU"], {x, -12, 12}, PlotLabel -> "SoftPlus", AxesLabel -> {None,
Style["f(x)", Italic]}, FrameLabel -> {{None, None}, {Style["x", Bold, 12],
None}}, PlotStyle -> ColorData[97, 25], Frame -> True], Plot[Log[Exp[x]
+ 1], {x, -12, 12}, PlotLabel -> "Log[Exp[x]+1]", AxesLabel -> {None,
Style["f(x)", Italic]}, FrameLabel -> {{None, None}, {Style["x", Bold, 12],
None}}, PlotStyle -> ColorData[97, 25], Frame -> True]}
Out[25]=

```



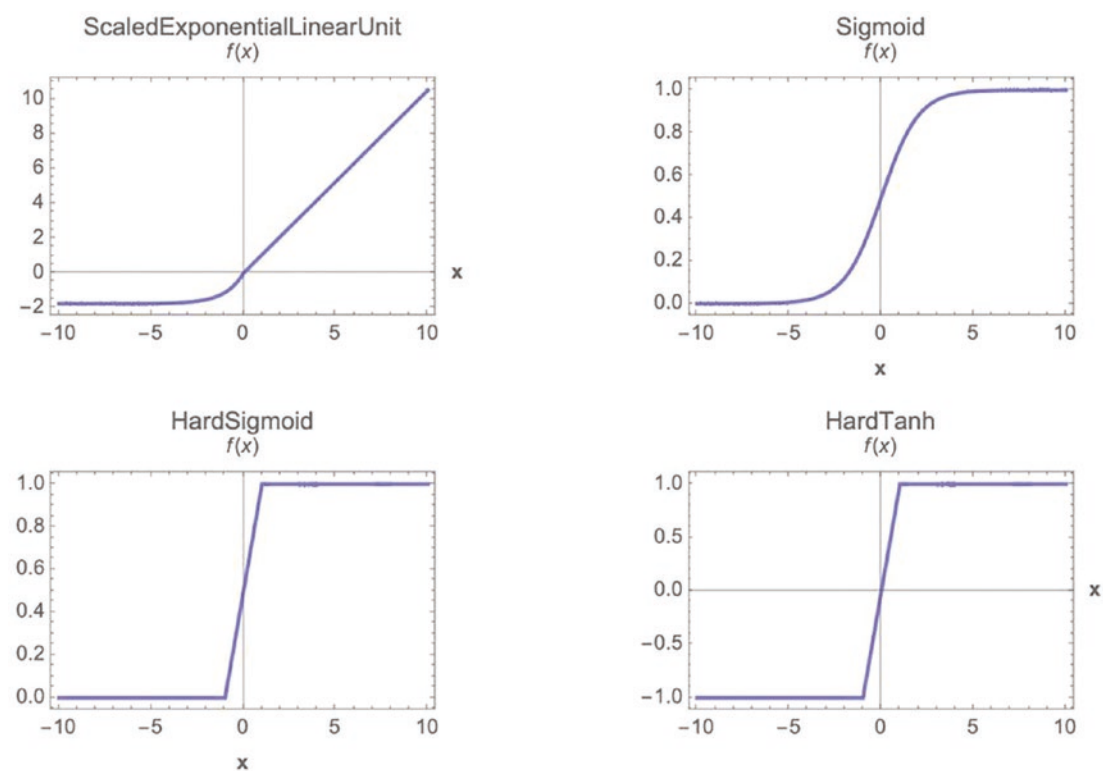
**Figure 9-15.** *SoftPlus function generated by the associated name and pure function*

Other standard functions are shown in the next plots, such as the scaled exponential linear unit, sigmoid, hard sigmoid, and hard hyperbolic tangent (see Figure 9-16). To view the functions supported, visit the documentation and type `ElementwiseLayer` in the search box.

```

In[26]:= GraphicsGrid@Partition[Table[If[Or[activation == "Sigmoid",
activation == "HardSigmoid"], Plot[ElementwiseLayer[activation]
[x, TargetDevice -> "CPU"], {x, -10, 10}, FrameLabel -> {Style["x",
Bold], None}, AxesLabel -> {None, Style["f(x)", Italic]}, PlotStyle
-> ColorData[97, 25], Frame -> True, PlotLabel -> activation],
Plot[ElementwiseLayer[activation][x, TargetDevice -> "CPU"], {x, -10, 10},
AxesLabel -> {Style["x", Bold], Style["f(x)", Italic]}, PlotStyle ->
ColorData[97, 25], Frame -> True, PlotLabel -> activation]], {activation,
{"ScaledExponentialLinearUnit", "Sigmoid", "HardSigmoid", "HardTanh"}}, 2]
Out[26]=

```



**Figure 9-16.** Plot of four different activation functions

## Softmax Layer

SoftmaxLayer is a layer that uses the expression  $S(x_i) = \frac{Exp(x_i)}{\sum_{j=1}^n Exp(x_j)}$ , where  $x$  represents a vector and  $x_i$  the components of the vector. This expression is known as the Softmax function. The functionality of this layer consists of converting a vector to a normalized vector, which consists of values in the range of 0 to 1. This layer is generally used to represent a partition of the classes based on the probabilities of each one, and it is used for tasks that involve classification. The input and output forms in the SoftmaxLayer can be entered as the other common layers except for the shape of “Real.”

```
In[27]:= sFL=SoftmaxLayer["Input"-> 4,"Output"-> 4];
```

Now, the layer can be applied to data.

```
In[28]:= SetAccuracy[sFL[{9,8,7,6}],3]
Out[28]= {0.64,0.24,0.09,0.03}
```

The total of the latter equals 1. `SoftmaxLayer` allows you to specify the level depth of normalization, which is seen in the parameter's properties of the layer. A level of -1 produces the normalization of a flattened list. Also, `SoftmaxLayer` can receive multidimensional arrays, not just flattened lists.

```
In[29]:= SoftmaxLayer[1,"Input"->{3,2}];
SetPrecision[%[{7,8},{8,7},{7,8}]],3]//MatrixForm
Out[30]//MatrixForm=
```

$$\begin{pmatrix} 0.212 & 0.422 \\ 0.576 & 0.155 \\ 0.212 & 0.422 \end{pmatrix}$$

Summing the elements of the first columns gives the same for the second column. Another practical layer is called `CrossEntropyLossLayer`. This layer is widely used as a loss function for classification tasks. This loss layer measures how well the classification model performs. Entering the string `Probabilities` as an argument of the loss layer computes the cross-entropy loss by comparing the input class probability to the target class probability.

```
In[31]:= CrossEntropyLossLayer["Probabilities","Input"->3 ];
```

Now, the target form is set to the probabilities of the classes; the inputs and targets are entered the same way as with `MeansSquaredLoss`.

```
In[32]:= %[<|"Input"->{0.2,0.5,0.3},"Target"->{0.3,0.5,0.2}|>]
Out[32]= 1.0702
```

Setting the `Binary` argument in the layer is used when the probabilities constitute a binary alternative.

```
In[33]:= CrossEntropyLossLayer["Binary","Input"-> 1];
%[<|"Input"-> 0.1,"Target"-> 0.9|>]
Out[34]= 2.08286
```

To summarize the properties of layers in the Wolfram Language, the inputs and outputs of the layers are always scalars and numeric matrixes. Layers are evaluated using lower number precision, such as single-precision numbers. Layers have the property of being differentiable; this helps the model to perform efficient learning since some

learning methods go into convex optimization problems. The Wolfram Language has many layers, each with specific functions. To display all the layers within Mathematica, check the documentation or write `?* Layer`, which gives you the commands with the word `layer` associated at the end. Each layer has different behaviors, operations, and parameters, although some may resemble other commands, such as `Append` and `AppendLayer`. It is important to know the different layers and what they can do to best use them.

## Function Layer

Another recently introduced (version 12.2) and updated (version 13) layer is the `FunctionLayer`. Unlike the `ElementwiseLayer`, this layer allows users to apply custom functions that do not come by default in the documentation library. This makes it a flexible tool for more complex operations, where the function to be applied is determined by the user (see Figure 9-17).

```
In[35]:= FunctionLayer[#*4&]
Out[35]=
```



**Figure 9-17.** A function layer that multiplies the input (#) by 4, and & is the pure function

The input and output definitions are similar to the previous layers you have seen. It can be an arbitrary array of input with no shape specification. However, the output shape is determined based on the function used within the layer; for instance, in the previous example, the input is a scalar (represented as a one-element array) and returns a scalar.

```
In[36]:= FunctionLayer[1/(1+Exp[-#])&];
%[{2,-3,4}]
Out[37]= {0.880797,0.0474259,0.982014}
```

With `FunctionLayer`, built-in functions can also be used instead of user-defined functions, for instance, the logistic sigmoid function, which returns the same as the latter code.



```
In[38]:= FunctionLayer[LogisticSigmoid];
%[{2,-3,4}]
Out[39]= {0.880797,0.0474259,0.982014}
```

A difference between `FunctionLayer` and `ElemewiseLayer` is that you can apply a function to each element independently in the first. On the other hand, it performs element-wise operations, ensuring shape consistency.

## Encoder and Decoders

Suppose audio, images, or other types of variables are intended to be used. In that case, this type of data needs to be converted into a numeric array to be introduced as input into a layer. This is where encoders and decoders come into play.

### Encoder

Layers must have a `NetEncoder` attached to the input to perform a correct construction. The `NetEncoders` interpret the image, audio, and data to a numeric value to be used inside a net model. Different names are associated with the encoding type. The most common are `Boolean` (True or False, encoding as 1 or 0), `Characters` (string characters as one-hot vector encoding), `Class` (class labels as integer encoding), `Function` (custom function encoding), `Image` (2D image encoding as a rank 3 array), and `Image3D` (3D image encoding as a rank 4 array). The arguments of the encoder are the name or the name and the corresponding features of the encoder (see Figure 9-18).

```
In[40]:= NetEncoder["Boolean"]
Out[40]=
```

NetEncoder [  Type: Boolean  
Dimensions: {} (scalar)  
Output: boolean ]

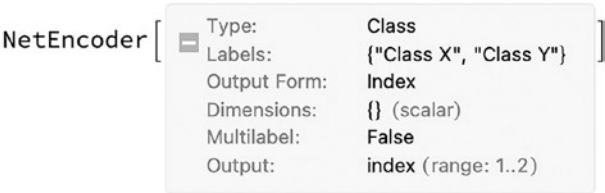
**Figure 9-18.** Boolean type `NetEncoder` To test the encoder, you use the following.

To test the encoder, you use the following.

```
In[41]:= Print["Booleans:",{ %[True], %[False] }]
Out[40]= Booleans:{1,0}
```

A NetEncoder can have classes with different index labels. Like a classification of class X and class Y, this corresponds to an index of the range from 1 to 2 (see Figure 9-19).

```
In[42]:= NetEncoder[{"Class",{"Class X","Class Y"}}]
Out[42]=
```



**Figure 9-19.** Class type NetEncoder

```
In[43]:= Print["Classes:", %Table[RandomChoice[{"Class X", "Class Y"}],
{i, 10}]]]
Out[43]= Classes:{1,1,2,2,2,2,1,1,1}
```

The following is used for a unit vector.

```
In[44]:= NetEncoder[{"Class",{"Class X","Class Y","Class Z"},
"UnitVector"}]; Print["Unit Vector:",%Table[RandomChoice[{"Class X",
"Class Y","Class Z"}],{i,5}]]] Print["MatrixForm:",%Table[RandomChoice[{"
Class X","Class Y","Class Z"}],{i,5}]]//MatrixForm[#]&]
Out[47]= Unit Vector:{0,1,0},{0,1,0},{0,1,0},{1,0,0},{0,0,1}}
```

MatrixForm: 
$$\begin{pmatrix} 0 & 0 & 1 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \\ 1 & 0 & 0 \end{pmatrix}$$

Depending on the name used inside NetEncoder, properties related to the encoder may vary. This is depicted in the different encoder objects that are created. To attach a NetEncoder to a layer, the encoders are entered at the input port—for example, for an ElementwiseLayer (see Figure 9-20). In this case, the input port of the layer has the name Boolean; the layer recognizes that this is a NetEncoder of a Boolean type. Clicking the name Boolean shows the relevant properties.

```
In[47]:= ElementwiseLayer[Sin,"Input"->NetEncoder["Boolean"]]
Out[47]=
```

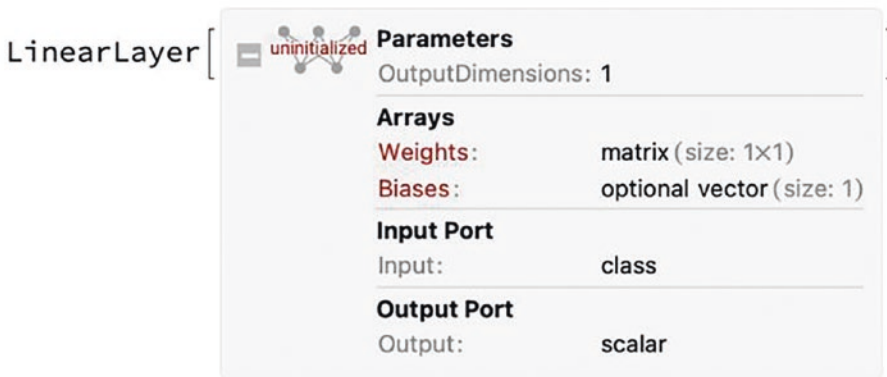


**Figure 9-20.** Layer with an encoder attached to the input port

For a LinearLayer, use the following form.

```
In[48]:= LinearLayer["Input"->NetEncoder[{"Class",{ "Class X","Class Y"}]],
"Output"->"Scalar"]
Out[48]=
```

Clicking the input port shows the encoder specifications, as Figure 9-21 shows.



**Figure 9-21.** Class encoder attached to a Linear Layer

A NetEncoder is also used to convert images into numeric matrixes or arrays by specifying the class, the size or width, and height of the output dimensions, and the color space, which can be grayscale, RGB, CMYK, or HSB (hue, saturation, and brightness); for example, encoding an image that produces a  $1 \times 28 \times 28$  array in grayscale, or  $3 \times 28 \times 28$  array in an RGB scale (see Figure 9-22), no matter the size of the input image. The first rank of the array represents the color channel, and the other two represent the spatial dimensions.

```
In[49]:= Table[NetEncoder[{"Image",{28,28},"ColorSpace"->Color}],
{Color,{"Grayscale","RGB"}}]
Out[49]=
```



**Figure 9-22.** *NetEncoders for grayscale and RGB scale images*

Once the encoder has been established, it can be applied to the desired image; then, the encoder creates a numeric matrix with the specified size. Creating a NetEncoder for an image shows relevant properties such as type, input image size, and color space, among others. Applying the encoder generates a matrix in the size previously established.

```
In[50]:=I imgEncoder = NetEncoder[{"Image", {3, 3}, "ColorSpace" ->
"CMYK"}]; Print["Numeric Matrix:", SetPrecision[%[ExampleData[{"TestImage",
"House"}]]], 3] // MatrixForm]
Out[50]=
```

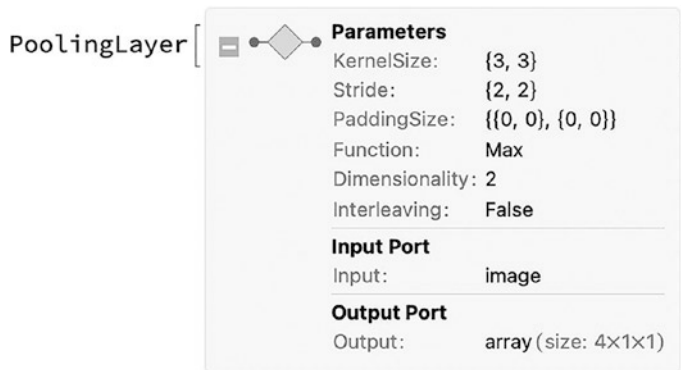
$$\begin{pmatrix} \begin{pmatrix} 0.255 \\ 0.168 \\ 0.255 \end{pmatrix} \begin{pmatrix} 0.145 \\ 0.00392 \\ 0.255 \end{pmatrix} \begin{pmatrix} 0.0784 \\ 0.0116 \\ 0.0274 \end{pmatrix} \\ \begin{pmatrix} 0.153 \\ 0.196 \\ 0.129 \end{pmatrix} \begin{pmatrix} 0.2 \\ 0.31 \\ 0.255 \end{pmatrix} \begin{pmatrix} 0.259 \\ 0.349 \\ 0.306 \end{pmatrix} \\ \begin{pmatrix} 0.047 \\ 0.102 \\ 0.00784 \end{pmatrix} \begin{pmatrix} 0.164 \\ 0.321 \\ 0.164 \end{pmatrix} \begin{pmatrix} 0.262 \\ 0.384 \\ 0.146 \end{pmatrix} \\ \begin{pmatrix} 0.16 \\ 0.262 \\ 0.184 \end{pmatrix} \begin{pmatrix} 0.255 \\ 0.408 \\ 0.478 \end{pmatrix} \begin{pmatrix} 0.325 \\ 0.388 \\ 0.569 \end{pmatrix} \end{pmatrix}$$

The output generated is a numeric matrix that is now ready to be implemented in a network model. If the input image shape is in a different color space, the encoder reshapes and transforms the image into the established color space. The image used in this example is obtained from the ExampleData[{"TestImage," "House"}].

## Pooling Layer

Encoders can be added to the ports of single layers or containers by specifying the encoder to the port—for instance, a `PoolingLayer`. These layers are used primarily on convolutional neural networks (see Figure 9-23).

```
In[52]:= poolLayer=PoolingLayer[{3,3},{2,2},PaddingSize->0,"Function"->
Max,"Input"-> NetEncoder[{"Image",{3,3},"ColorSpace"-> "CMYK"}](*Or
ImgEncoder*)]
Out[52]=
```



**Figure 9-23.** *PoolingLayer with a NetEncoder*

The latter layer has a specification for a two-dimensional `PoolingLayer` with a kernel size of  $3 \times 3$  and a stride of  $2 \times 2$ , which is the step size between kernel applications. `PaddingSize` adds elements at the beginning and the end of the input matrix. This is done so that the division between the matrix and kernel sizes is an integer, preventing the loss of information between layers. `Function` indicates the pooling operation function, which is `Max`; this calculates the maximum value in each filter patch. It can also compute the mean and total for the average and summation of the filter values, respectively. Sometimes, they might be known as max, average, and sum pooling layers.

```
In[53]:=SetPrecision[poolLayer[ExampleData[{"TestImage","House"}]],3]
//MatrixForm
Out[53]//MatrixForm=
```

$$\begin{pmatrix} (0.255) \\ (0.349) \\ (0.384) \\ (0.569) \end{pmatrix}$$

## Decoders

Once the net operations are finished, it return numeric expressions. On the other hand, in some tasks, you do not want numeric expressions, such as in classification tasks where classes can be given as outputs, where the model can tell that a particular object belongs to a class A and another object belongs to a class B, so a vector or numeric array can represent a probability of each class. To convert the numeric arrays into other forms of data, a NetDecoder is used (see Figure 9-24).

```
In[54]:= decoder=NetDecoder[{"Class",CharacterRange["W","Z"]}]
Out[54]=
```



**Figure 9-24.** NetDecoder for four different classes

The dimension of the decoder is equal to class construction. You can apply a vector of probabilities, and the decoder interprets it and tells you the class to which it belongs. It also displays the probabilities of the classes.

```
In[55]:= decoder@{0.3,0.2,0.1,0.4}(*This is the same as Decoder[{0.3,0.2,0.1,0.4},
1,0,4},"Decision"] *)
Out[55]= Z
```

TopDecisions, TopProbabilites, and uncertainty of the probability distribution are displayed as follows.

```
In[56]:= TableForm[{decoder[{0.3, 0.2, 0.1, 0.4},
"TopDecisions" -> 4](* or {"TopDecisions", 4} the same is for
TopProbabilities*), decoder[{0.3, 0.2, 0.1, 0.4}, "TopProbabilities"
```

```
-> 4], decoder[{0.3, 0.2, 0.1, 0.4}, "Entropy"]], TableDirections
-> Column, TableHeadings -> {{Style["TopDecisions", Italic],
Style["TopProbabilities", Italic], Style["Entropy", Italic]},
None}]Out[56]//TableForm=
```

TopDecisions	Z	W	X	Y
TopProbabilities	Z->0.4	W->0.3	X->0.2	Y->0.1
Entropy	1.27985			

Given the list of values, input depth is added to define the class's application level.

```
In[57]:= NetDecoder[{"Class",CharacterRange["X","Z"],"InputDepth"→2}];
```

Applying the decoder to a nested list of values produces the following.

```
In[58]:= TableForm[%[{{0.1, 0.3, 0.6}, {0.3, 0.4, 0.3}}, "TopDecisions" ->
3](* or {"TopDecisions", 4} the same is for TopProbabilities*), %[{{0.1,
0.3, 0.6}, {0.3, 0.4, 0.3}}, "TopProbabilities" -> 3], %[{{0.1,
0.3, 0.6}, {0.3, 0.4, 0.3}}, "Entropy"]], TableDirections ->
Column, TableHeadings -> {{Style["TopDecisions", Italic],
Style["TopProbabilities", Italic], Style["Entropy", Italic]}, None}]
Out[58]//TableForm=
```

TopDecisions	Z	Y
	Y	X
	X	Z
	Z->0.6	Y->0.4
TopProbabilities	Y->0.3	X->0.3
	X->0.1	Z->0.3
Entropy	0.897946	1.0889

A decoder is added to the output port of a layer, container, or network model.

```
In[59]:= SoftmaxLayer["Output"→NetDecoder[{"Class",{ "X","Y","Z"}]]];
```

Applying the layer to the data produce the probabilities for each class.

```
In[60]:= {#@{1,3,5},%[{1,3,5},"Probabilities"],%[{1,3,5},"Decision"]}
Out[60]= {Z,<|X->0.0158762,Y->0.11731,Z->0.866813|>,Z}
```

## Applying Encoder and Decoders

You are ready to implement the whole process of encoding and decoding in Figure 9-25. First, the image is resized by 200 pixels in width to show how the original image looks before encoding.

```
In[61]:= Img=ImageResize[ExampleData[{"TestImage","House"}],200]
Out[61]=
```



**Figure 9-25.** Example image of a house when the encoder and decoder are defined

```
In[62]:= encoder=NetEncoder[{"Image",{100,100},"ColorSpace"-> "RGB"}];
decoder=NetDecoder[{"Image",ColorSpace-> "Grayscale"}];
```

Then, the encoder is applied to the image, and the decoder is applied to the numeric matrix. The dimensions of the decoded image are checked to see if they match the encoder output dimensions (see Figure 9-26).

```
In[64]:=encoder[img];
decoder[%]
```





**Figure 9-26.** *Example of the decoded house*

Figure 9-26 shows that the image has been converted into a grayscale image with new dimensions.

```
In[66]:= ImageDimensions[%]
Out[66]= {100,100}
```

As seen, the picture has been resized. Try to look at the steps in the process, like viewing the numeric matrix and the objects corresponding to the encoder and decoder. Using the encoders and decoders involves the data type you use because every net model receives different inputs and generates different outputs.

## NetChains and Graphs

Neural networks consist of different layers, not individual layers on their own. The NetChain command or the NetGraph command is used to construct more complex structures with more than one layer.

## Containers

Containers are valuable for properly operating and constructing neural networks in the Wolfram Language. In the Wolfram Language, containers are structures that assemble the infrastructure of the neural network model. Containers can have multiple forms. NetChain is useful for creating linear and non-linear structures' nets. This helps the model to learn non-linear patterns. You can think that each layer in a network has a level of abstraction that detects complex behavior, which could not be recognized if you

only worked with one single layer. As a result, you can build networks in a general way, starting from three layers: the input layer, the hidden layer, and the output layer. When there are more than two hidden layers, it is deep learning; for more information, refer to *Introduction to Deep Learning: From Logical Calculus to Artificial Intelligence* by Sandro Skansi (Springer, 2018).

NetChain can join two operations. They can be written as a pure function instead of just the function's name (see Figure 9-27).

```
In[67]:=NetChain[{ElementwiseLayer[LogisticSigmoid@#&],ElementwiseLayer[Sin@#&]}]
Out[67]=
```



**Figure 9-27.** NetChain containing two elementwise layers

The object returned is a NetChain, and the icon of three colored rectangles appears. This means that the object created (NetChain) or referred to is a net chain and contains layers. If the chain is examined, it shows the input, first (LogisticSigmoid), second (Sin), and output layers. The operations are in order of appearance, so the first layer is applied and then the second. The input and output options of other layers are supported in NetChain, such as a single real number (Real), an integer (Integer), an “n”-length vector, and a multidimensional array (see Figure 9-28).

```
In[68]:= NetInitialize@NetChain[{3,4,12,Tanh}, "Input" ->1]
Out[68]=
```



**Figure 9-28.** NetChain with multiple layers

NetChain recognizes the Wolfram Language function names and associates them with their corresponding layers, like 3, 4, and 12. They represent a linear layer with outputs of sizes 3, 4, and 12 (see Figure 9-28). The Tanh function represents the elementwise layer.

Let's append a layer to the chain created with NetAppend (see Figure 9-29) or NetPrepend. Many of the original commands of the Wolfram Language have the same meaning—for example, to delete in a chain would be NetDelete[net\_name, #\_of\_layer].

```
In[69]:=NetInitialize@NetChain[{1,ElementwiseLayer[LogisticSigmoid@#&]], "Input" -> 1];
netCH2=NetInitialize@NetAppend[%,{1,ElementwiseLayer[Cos[#]&]}}]
Out[70]=
```



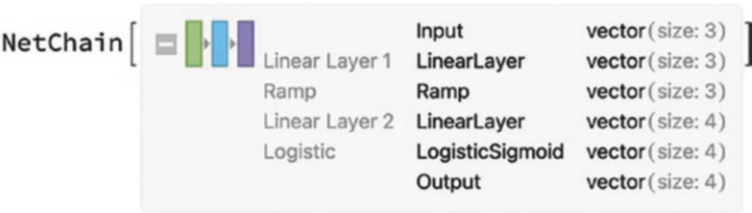
**Figure 9-29.** NetChain object with different added layers

Different options are available when a net is applied to data, such as NetEvaluationMode (mode of evaluation, either train or test), TargetDevice, and WorkingPrecision (numeric precision).

```
In[71]:= netCH2[{{0},{2},{44}},NetEvaluationMode-> "Train",TargetDevice->
"CPU",WorkingPrecision-> "Real64",RandomSeeding-> 8888](*use N@Cos[Sin[Logi
sticSigmoid[{0,2,44}]]] to check results*)
Out[71]= {{0.967873},{0.990894},{1.}}
```

Another form is to enter the explicit names of layers in a chain, which is typed as an association (see Figure 9-30).

```
In[72]:= NetInitialize@NetChain[<|"Linear Layer 1"->LinearLayer[3],
"Ramp"-> Ramp,"Linear Layer 2"->LinearLayer[4],"Logistic"-> ElementwiseLayer
[LogisticSigmoid]|>,"Input"-> 3]
Out[72]=
```



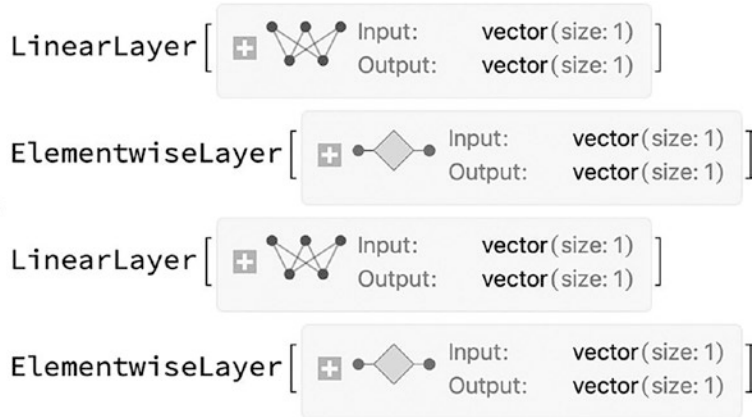
**Figure 9-30.** *NetChain object with custom layer names*

Inspecting the layer's contents should appear after clicking the layer's name or the layer. If a layer wants to be extracted, then NetExtract is used along with the name of the corresponding layer. The output is suppressed, but the layer should pop out if the semicolon is removed.

```
In[73]:=NetExtract[%, "Logistic"];
```

To extract all of the layers in one line of code, Normal does the job (see Figure 9-31).

```
In[74]:= Normal[netCH2]//Column
Out[74]=
```



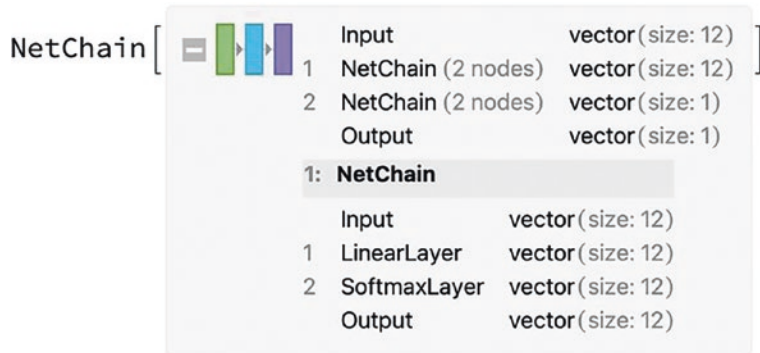
**Figure 9-31.** *Layers of the NetChain NetCH2*

## Multiple Chains

Chains can be joined with a nested chain (see Figure 9-32).

```
In[75]:= chain1=NetChain[{12, SoftmaxLayer[]}];
chain2=NetChain[{1, ElementwiseLayer[Cos[#]&]}];
```

```
nestedChain=NetInitialize@NetChain[{chain1,chain2},"Input"-> 12]
Out[77]=
```



**Figure 9-32.** Chain 1 selected of the two chains available

This chain is divided into two NetChains, each representing a chain. In this case, you see chain1 and chain2, and each chain shows its corresponding nodes. To flatten the chains, use NetFlatten (see Figure 9-33).

```
In[78]:= NetFlatten[nestedChain]
Out[78]=
```

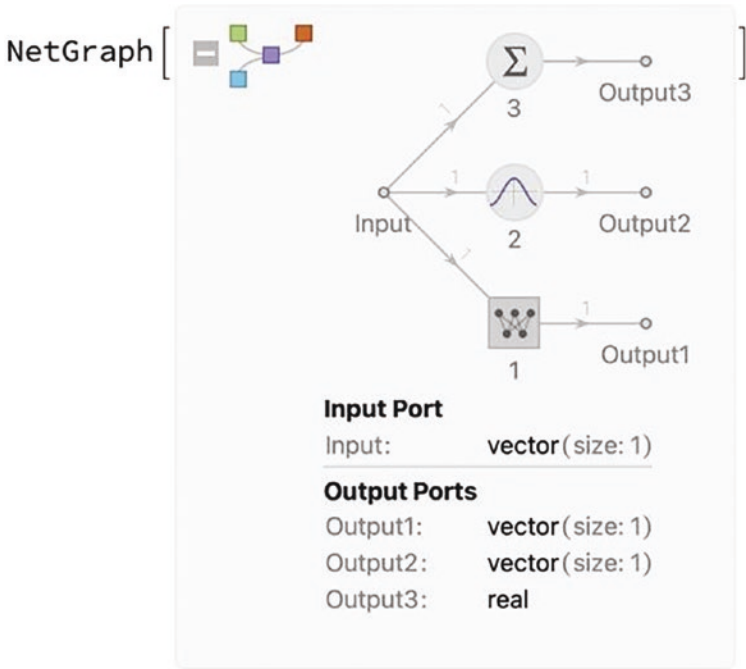


**Figure 9-33.** Flattened chain

## NetGraphs

The NetChain command only joins layers in which the output of a layer is connected to the input of the next layer. NetChain does not work in connecting inputs or outputs to other layers; it only works with one layer. To work around this, the use of NetGraph is required. Besides allowing more inputs and layers, NetGraph represents the neural network's structure and process with a graph (see Figure 9-34).

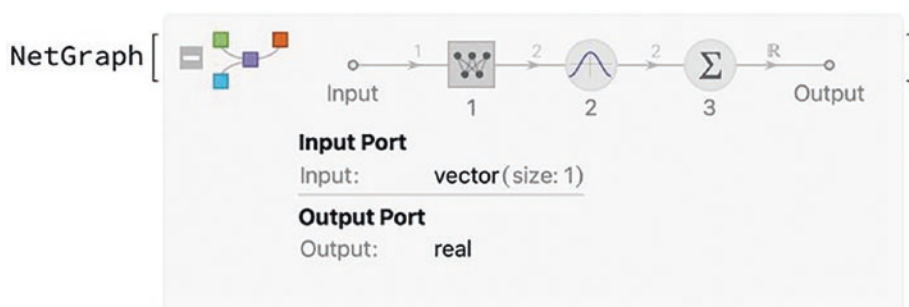
```
In[79]:= NetInitialize@NetGraph[{ LinearLayer["Output"-> 1,"Input"-> 1],
Cos,SummationLayer[]},{},{}]
Out[79]=
```



**Figure 9-34.** Expanded NetGraph

The object crafted is a NetGraph, represented by the figure of the connecting squares, as seen in Figure 9-35. The input goes to three different layers, each with its output. NetGraph accepts two arguments: the first is for the layers or chains, and the second is to define the graph vertices or connectivity of the net. For example, the net has three outputs in the latter code because the vertices were not specified. SummationLayer is a layer that sums all the input data.

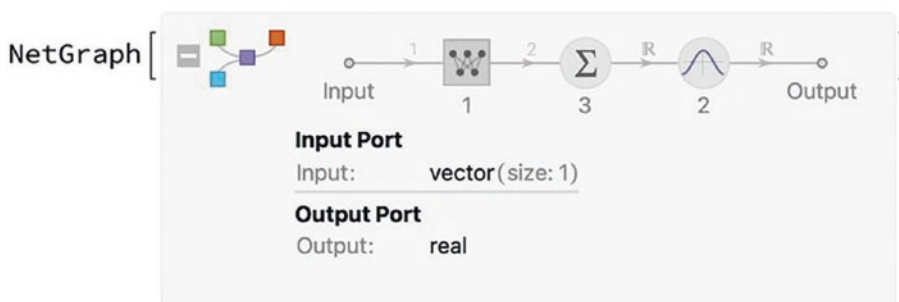
```
In[80]:= net1=NetInitialize@NetGraph[{ LinearLayer["Output"-> 2,"Input"->
1],Cos,SummationLayer[]},{1-> 2-> 3}]
Out[80]=
```



**Figure 9-35.** *Unidirectional NetGraph*

The vertex notation means that the output of a layer is given to another layer, and so on. In other words,  $1 \rightarrow 2 \rightarrow 3$  means that the output of the linear layer is passed to the next layer until it is finally summed up in the last layer with the summation layer (see Figure 9-35), thus preserving the order of appearance of the layers. However, you can alter the order of each vertex. The net can be modified so that outputs can go to other layers of the net, such as 1 to 3 and then to 2 (see Figure 9-36). With NetGraph, layers and chains can be entered as a list or an association. The vertices are typed as a list of rules.

```
In[81]:= net2=NetInitialize@NetGraph[{ LinearLayer["Output"-> 2,"Input"->
1],Cos,SummationLayer[]},{1-> 3->2}]
Out[81]=
```

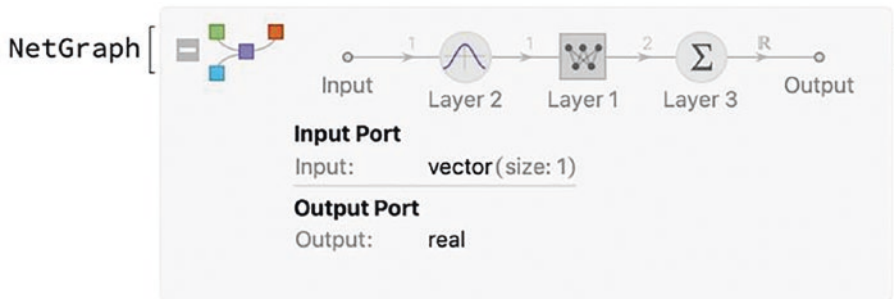


**Figure 9-36.** *NetGraph* structure of *Net2*

The inputs and outputs of each layer are marked by a tooltip that appears when passing the cursor over the graph lines or vertices. Because input and output are not specified, NetGraph infers the data type in the input and output port; this is the case for the capital R in the input and output of the layer used, which stands for real.

With NetGraph, layers can be entered as a list or association. The connections are typed as a list of rules (see Figure 9-37).

```
In[82]:= NetInitialize@NetGraph[<|"Layer 1"-> LinearLayer[2,"Input"->
1],"Layer 2"-> Cos,"Layer 3"-> SummationLayer[]|>,{ "Layer 2"-> "Layer 1"->
"Layer 3"}]
Out[82]=
```

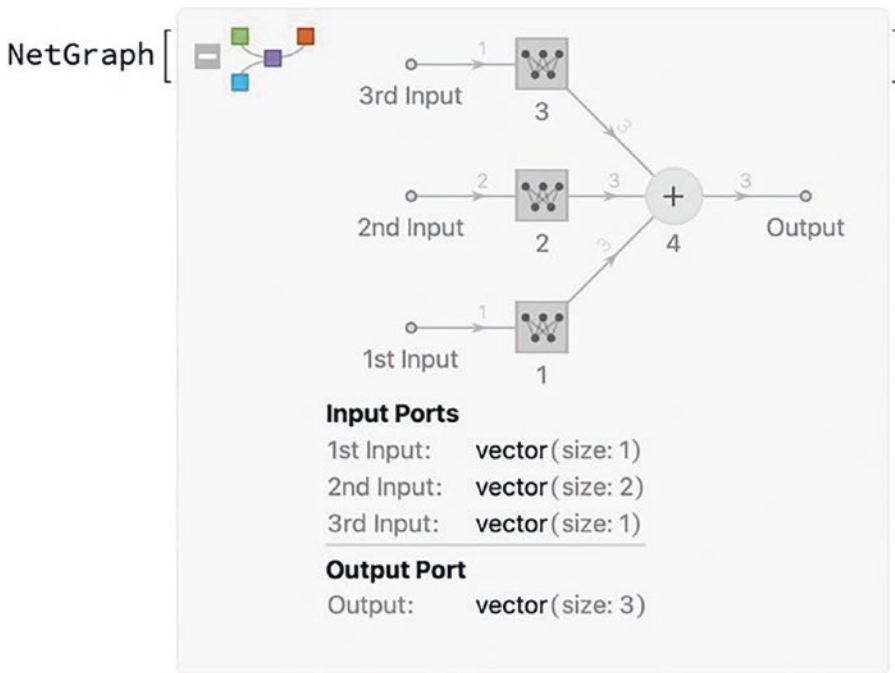


**Figure 9-37.** NetGraph initialized with named layers

It is possible to specify how many inputs and outputs a structure can have from the NetPort command (see Figure 9-38).

```
In[83]:= NetInitialize@ NetGraph[{ LinearLayer[3, "Input" ->
1], LinearLayer[3, "Input" -> 2], LinearLayer[3, "Input" -> 1] ,
TotalLayer[]}, {NetPort["1st Input"] -> 1, NetPort["2nd Input"] ->
2, NetPort["3rd Input"] -> 3, {1, 2, 3} -> 4}] (*Or NetInitialize@
NetGraph[<|"L1"\[Rule] LinearLayer[3,"Input"\[Rule] 1],"L2"\[Rule]
LinearLayer[3,"Input"\[Rule] 1], "L3"\[Rule] LinearLayer[3,"Input"
\[Rule] 1] ,"Tot L"\[Rule] TotalLayer[]|>,{NetPort["1st Input"]\
\[Rule] "L1", NetPort["2nd Input"]\[Rule] "L2",NetPort["3rd Input"]\
\[Rule]"L3",{ "L1","L2","L3"} -> "Tot L"}]*)
Out[83]=
```





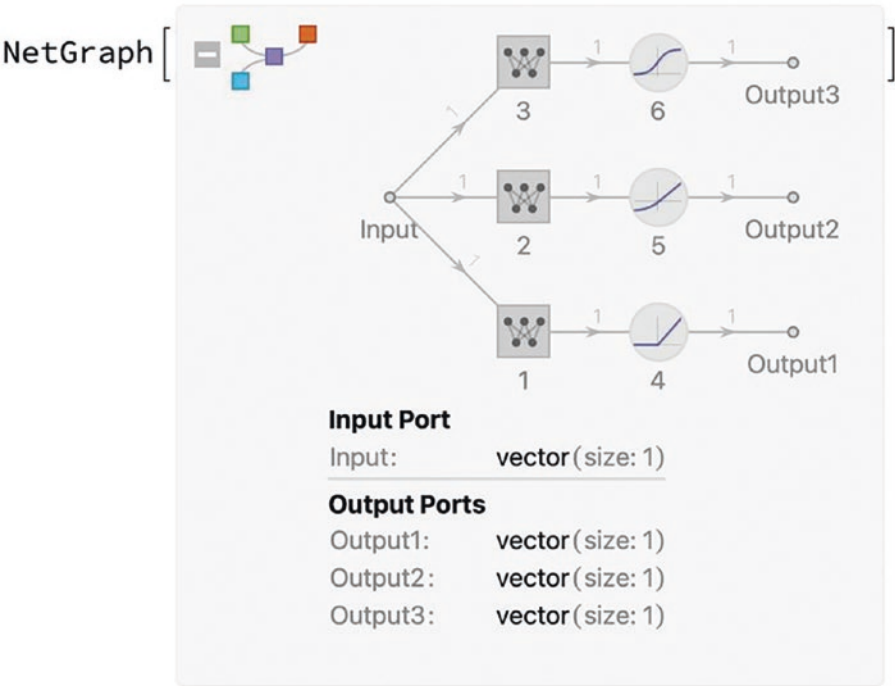
**Figure 9-38.** *NetGraph with multiple inputs and a single output*

If you have more than one input, each input is entered in the specified port.

```
In[84]:= %["1st Input"-> 32.32,"2nd Input"-> {2,\[Pi]},"3rd Input"-> 1|>]
Out[84]= {82.4758,-42.202,-37.4852}
```

If having more than one output, the results are displayed for every different output (see Figure 9-39).

```
In[85]:= NetInitialize[NetGraph[{LinearLayer[1,"Input"->
1],LinearLayer[1,"Input"-> 1],LinearLayer[1,"Input"-> 1],Ramp,El
ementwiseLayer["ExponentialLinearUnit"],LogisticSigmoid},{1->4->
NetPort["Output1"],2->5-> NetPort["Output2"],3-> 6-> NetPort["Output3"]}],
RandomSeeding->8888] %[{1}]
Out[85]=
```

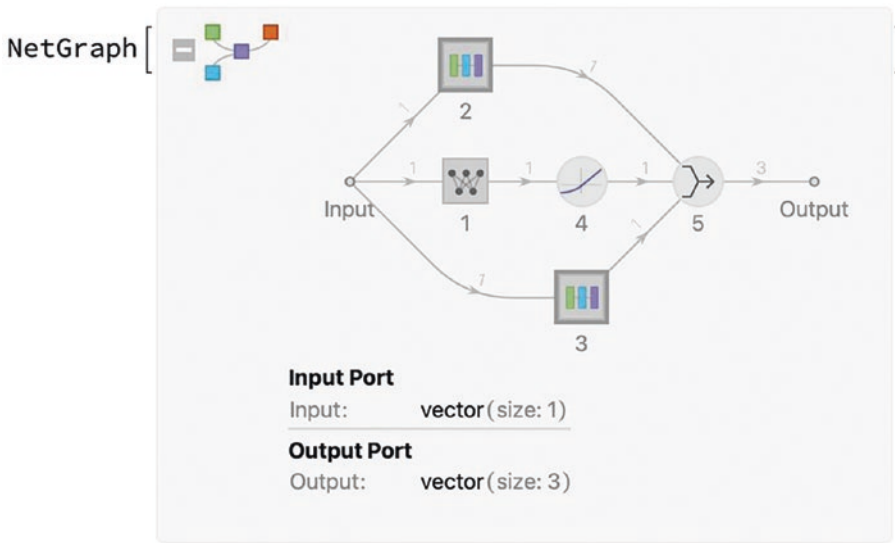


**Figure 9-39.** *NetGraph with single input and three outputs*

```
Out[86]= <|Output1->{0.},Output2->{-0.289052},Output3->{0.860635}|>
```

NetChain containers can be treated as layers with NetGraph (see Figure 9-40). Some layers, such as the CatenateLayer, accept zero arguments.

```
In[87]:= NetInitialize@NetGraph[{LinearLayer[1,"Input"-> 1], NetChain[{L
inearLayer[1,"Input"-> 1], ElementwiseLayer[LogisticSigmoid[#]&]}],NetCh
ain[{LinearLayer[1,"Input"-> 1],Ramp}], ElementwiseLayer["ExponentialLin
earUnit"]},
CatenateLayer[{}],{1->4,2->5,3-> 5,4-> 5}]
Out[87]=
```



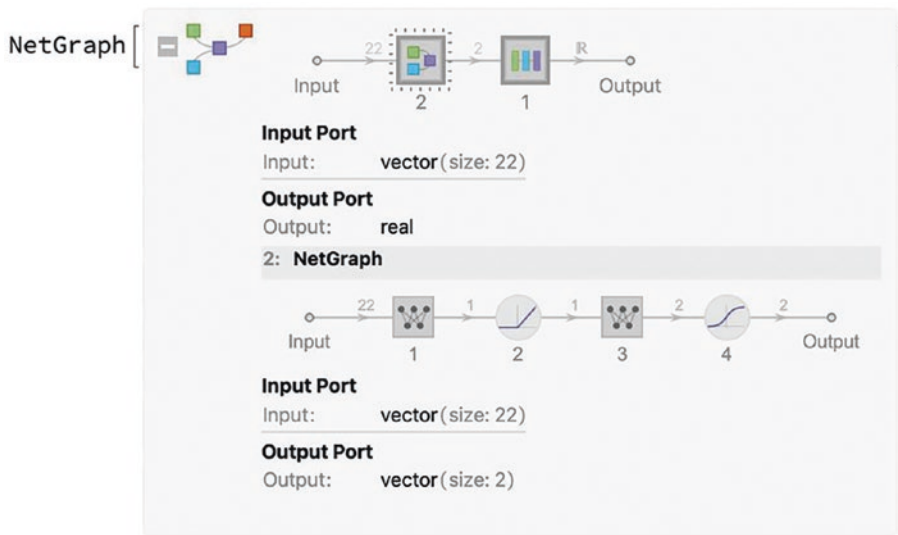
**Figure 9-40.** *NetGraph with multiple containers*

Clicking the chain or the layer shows the relevant information, and clicking the layer inside a chain gives the information about the layer on the selected chain.

## Combining Containers

NetChains, and NetGraphs can be nested to form different structures, as seen in the following example (see Figure 9-41), where a NetGraph and vice versa can follow a NetChain.

```
In[88]:= n1=NetGraph[{1,Ramp,2,LogisticSigmoid},{1-> 2,2-> 3,3-> 4}];
n2=NetChain[{3,SummationLayer[]}];
NetInitialize@NetGraph[{n2,n1},{2-> 1},"Input"-> 22]
Out[90]=
```



**Figure 9-41.** Nested NetGraph and NetChain

From the graph in Figure 9-40, it is clear that the input goes to the NetGraph, and the output of the NetGraph goes to the NetChain. A NetChain or NetGraph that has not been initialized appears in red. A fundamental quality of the containers (NetChain, NetGraph) is that they can behave as a layer. With this in mind, you can create nested containers involving only NetChains, NetGraphs, or both.

Just as a demonstration, more complex structures can be created with NetGraph, like those in Figure 9-42. Once a network structure is created, properties about every layer or chain can be extracted. For instance, with SummaryGraphic, you can obtain the graphic of the network graph.

```
In[91]:= net = NetInitialize@ NetGraph[{LinearLayer[10], Ramp, 10,
SoftmaxLayer[], TotalLayer[], ThreadingLayer[Times]], {1 -> 2 -> 3 -> 4,
{1, 2, 3} -> 5, {1, 5} -> 6}, "Input" -> "Real"];
Information[net, "SummaryGraphic"]
Out[92]=
```

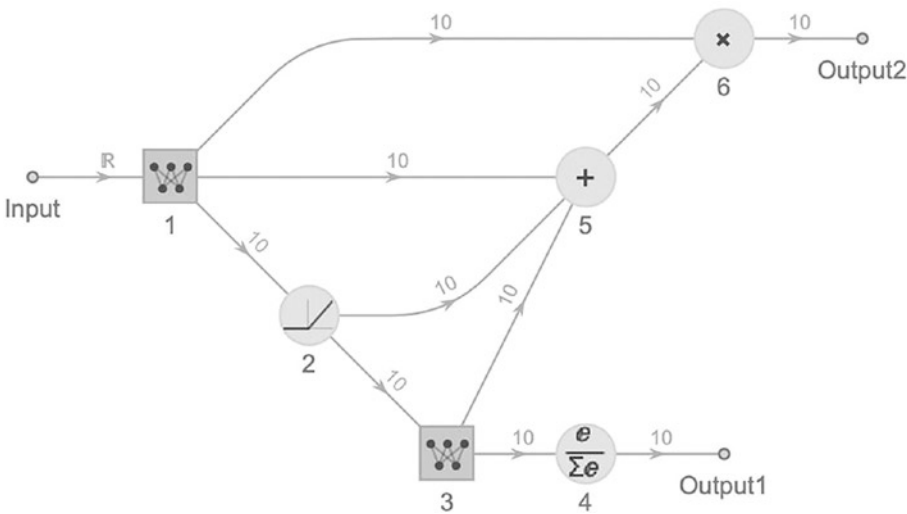


Figure 9-42. Compound graph net structure

## Network Properties

The properties related to the numeric arrays of the network are `Arrays` (gives each array in the network), `ArraysCount` (the number of arrays in the net), `ArraysDimensions` (dimensions of each array in the net), and `ArraysPositionList` (position of each array in the net), as depicted in Figure 9-43.

```
In[93]:={Dataset@Information[net,"Arrays"],Dataset@Information[net,"Arrays  
Dimensions"],Dataset@Information[net,"ArraysPositionList"]}  
Out[93]=
```

1	Biases	NumericArray[ <div>Type: Real32 Dimensions: {10}</div> ]		
1	Weights	NumericArray[ <div>Type: Real32 Dimensions: {10, 1}</div> ]		
3	Biases	NumericArray[ <div>Type: Real32 Dimensions: {10}</div> ]		
3	Weights	NumericArray[ <div>Type: Real32 Dimensions: {10, 10}</div> ]		

,

1	Biases	{10}	1	Biases
1	Weights	{10, 1}	1	Weights
3	Biases	{10}	3	Biases
3	Weights	{10, 10}	3	Weights

}

Figure 9-43. Datasat containing various properties

Information related to the variable type in the input and output ports are shown with `InputPorts` and `OutputPorts`.

```
In[94]:= {Information[net,"InputPorts"],Information[net,"OutputPorts"]}
Out[94]= {<|Input->Real|>,<|Output1->10,Output2->10|>}
```

You can see that the input is a real number, and the net has two output vectors of size 10. The most used properties related to layers are `Layers` (returns every layer of the net), `LayerTypeCounts` (number of occurrences of a layer in the net), `LayersCount` (number of layers in the net), `LayersList` (a list of all the layers in the net), and `LayerTypeCounts` (number of occurrences of a layer in the net). Figure 9-44 shows for `Layers` and `LayerTypeCounts`.

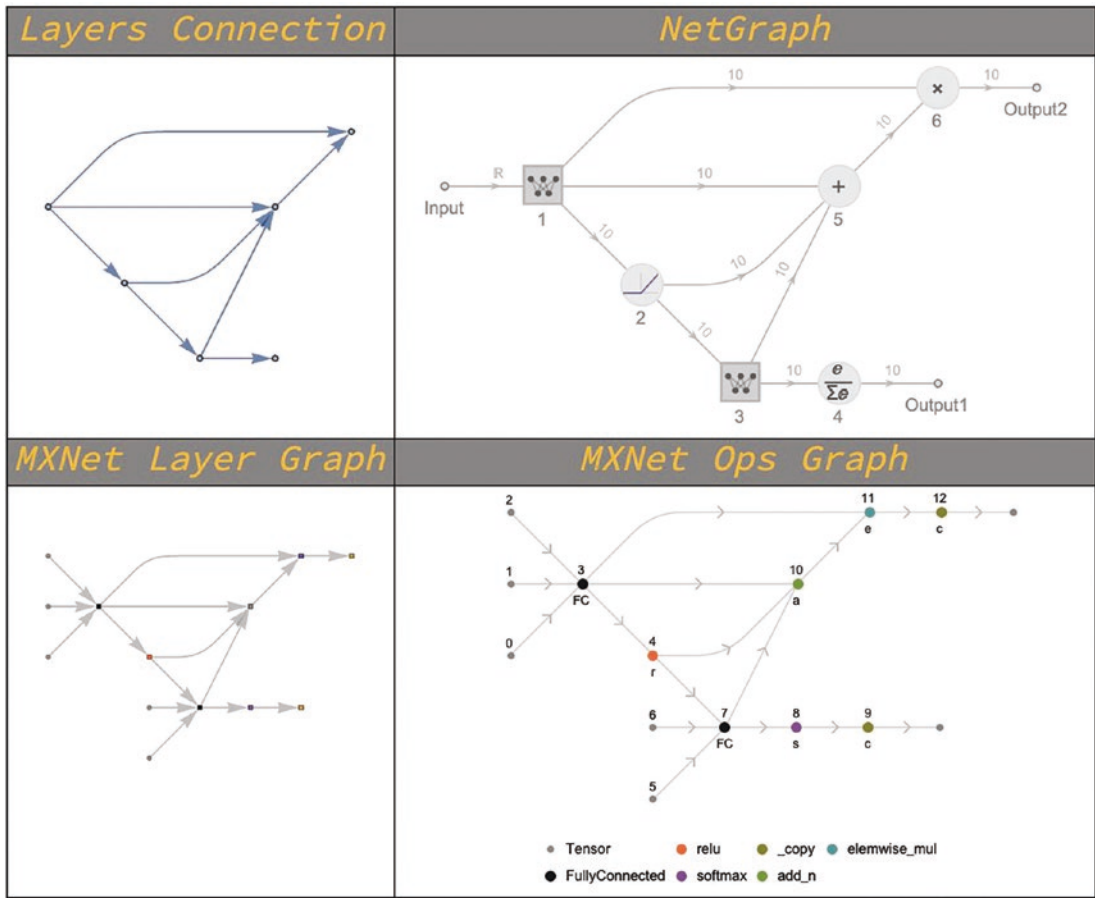
```
In[95]:=Dataset@{Information[net,"Layers"],Information[net,"LayerType
Counts"]}
Out[95]=
```

{1}	LinearLayer [ <div>Input: real Output: vector (size: 10)</div> ]
{2}	ElementwiseLayer [ <div>Input: vector (size: 10) Output: vector (size: 10)</div> ]
{3}	LinearLayer [ <div>Input: vector (size: 10) Output: vector (size: 10)</div> ]
{4}	SoftmaxLayer [ <div>Input: vector (size: 10) Output: vector (size: 10)</div> ]
{5}	TotalLayer [ <div><b>Input Ports</b> Input1: vector (size: 10) Input2: vector (size: 10) Input3: vector (size: 10) <b>Output Port</b> Output: vector (size: 10)</div> ]
{6}	ThreadingLayer [ <div><b>Input Ports</b> Input1: vector (size: 10) Input2: vector (size: 10) <b>Output Port</b> Output: vector (size: 10)</div> ]
LinearLayer	2
ElementwiseLayer	1
SoftmaxLayer	1
TotalLayer	1
ThreadingLayer	1

**Figure 9-44.** Information about the layers contained in the symbol Net

Visualization of the net structure (see Figure 9-45) is achieved with the properties LayersGraph (a graph showing the connectivity of the layers), SummaryGraphics (graphic of the net structure), MXNetNodeGraph (MXNeT raw graph operations), and MXNetNodeGraphPlot (annotated graph of MXNet operations). MXNet is an open-source deep learning framework that supports a variety of programming languages, and one of them is the Wolfram Language. In addition, the Wolfram Neural Network Framework works with MXNet structure as backend support.

```
In[96]:= Grid[{{Style["Layers Connection",Italic,20,ColorData[105,4]],Style
["NetGraph",Italic,20,ColorData[105,4]]},{Information[net,"LayersGraph"],In
formation[net,"SummaryGraphic"]},{Style["MXNet Layer Graph",Italic,20,Color
Data[105,4]],Style["MXNet Ops Graph",Italic,20,ColorData[105,4]]},{Informat
ion[net,"MXNetNodeGraph"],Information[net,"MXNetNodeGraphPlot"]}},Dividers-
>All,Background-> {{None,None}},{{Opacity[1,Gray],None}}}]
Out[96]=
```



**Figure 9-45.** Grid showing multiple graphics

Passing the cursor pointer over a layer or node in the MXNet symbol graph, a tooltip shows the properties of the MXNet symbols like ID, name, parameters, attributes, and inputs.



## Exporting and Importing a Model

Because of the interoperability of the Wolfram Language and MXNet, the Wolfram Language supports the import and export of neural nets, initialized or uninitialized. You create a folder on the desktop with the MXNet Nets name and export the network found in the Net variable.

```
In[97]:= fileDirectory="/Users/macosex/Desktop";
Export[FileNameJoin[{fileDirectory,"MxNet.json"}],net,"MXNet","ArrayPath"->
Automatic,"SaveArrays"-> True]
Out[98]= /Users/macosex/Desktop/MxNet.json
```

Exporting the network to the MXNet format generates two files: a JSON file that stores the topology of the neural network and a file of type .params that contains the required parameters (numeric arrays used in the network) data for the exported architecture; once it has been initialized. With ArrayPath set to Automatic, the params file is saved in the same net folder. Otherwise, it can have a different path. SaveArrays indicate whether the numeric arrays are exported (True) or not (False). Let's check the two files created in the MXNets Nets folder.

```
In[99]:= FileNames[All,File@fileDirectory]
Out[99]= {/Users/macosex/Desktop/MxNet.json,
/Users/macosex/Desktop/MxNet.params}
```

To import an MXNet network, the JSON and params files are recommended to be in the same folder because the Wolfram Language assumes that a certain JSON file matches the pattern of the params file. There are various ways to import a net, including Import[file\_name.json, "MXNet"] and Import[file\_name.json,{"MXNet," element}] (the same as with .param files). Since version 13, nets are no longer imported as net chains or net graphs but can now be imported as net external objects. However, if you don't intend to use the neural network outside of the Wolfram Language, it's much simpler to store it as a WLNet, which facilitates easier saving and retrieval within the Wolfram Language environment. To export the net to the WLNet format, you can use the following code: Export["file\_name.wlnet", <net\_symbol or variable\_name>]. Then, you can import the net using Import["file\_name.wlnet"]

```
In[100]:=Import[FileNameJoin[{fileDirectory,"MxNet.json"}],{"MXNet",
"NetExternalObject"},InputPorts-><|"Input"->{1}|>,"ArrayPath"->None];
```

The latter net was imported with the .params file automatically. To import the net without the parameters, use `ArrayPath` set to `None` or set the params file path. Importing the net parameters can be done with a list (`ArrayList`), the names (`ArrayNames`), or an association (`ArrayAssociation`), as shown in Figure 9-46.

```
In[101]:= Row[Dataset[Import[FileNameJoin[{fileDirectory,"MxNet.json"}],{"MXNet",#}]]&/@{"ArrayAssociation","ArrayList","ArrayNames"}]
Out[101]=
```

1.Biases	NumericArray [ Type: Real32 Dimensions: {10} ]	NumericArray [ Type: Real32 Dimensions: {10} ]	
1.Weights	NumericArray [ Type: Real32 Dimensions: {10, 1} ]	NumericArray [ Type: Real32 Dimensions: {10, 1} ]	1.Biases
3.Biases	NumericArray [ Type: Real32 Dimensions: {10} ]	NumericArray [ Type: Real32 Dimensions: {10} ]	1.Weights
3.Weights	NumericArray [ Type: Real32 Dimensions: {10, 10} ]	NumericArray [ Type: Real32 Dimensions: {10, 10} ]	3.Biases
			3.Weights

**Figure 9-46.** Different import options of the MXNet format

The elements of the net to import are `InputNames`, `NetExternalObject`, `NodeDataset` (a dataset of the nodes of the MXNet), `NodeGraph` (nodes graph of the MXNet), `NodeGraphPlot` (plot of nodes of the MXNet). The following dataset shows a few options listed before Figure 9-47.

```
In[102]:= {Import[FileNameJoin[{fileDirectory,"MxNet.json"}],{"MXNet","Node Dataset"}],Import[FileNameJoin[{fileDirectory,"MxNet.json"}],{"MXNet","NodeGraphPlot"}]}//Row
Out[102]=
```

	op	attrs	inputs
Input	null		
1.Weights	null		
1.Biases	null		
1	FullyConnected	2 total › 3 total ›	{0, 0, 0}
2\$0	relu		{3, 0, 0}
3.Weights	null		
3.Biases	null		
3	FullyConnected	2 total › 3 total ›	{4, 0, 0}
4\$0	softmax	1 total ›	{7, 0, 0}
Output1	_copy		{8, 0, 0}
5	add_n	1 total › 3 total ›	{3, 0, 0}
6\$0	elemwise_mul	2 total ›	{3, 0, 0}
Output2	_copy		{11, 0, 0}

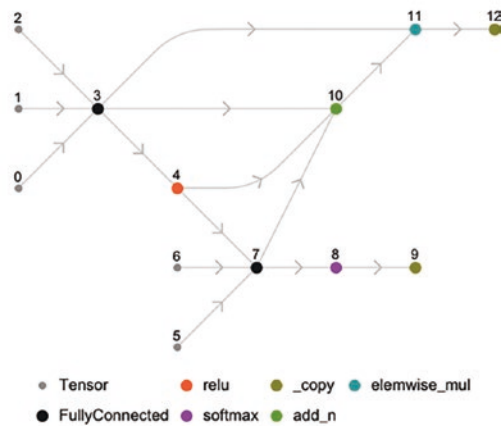


Figure 9-47. Node dataset and MXNet ops plot

Some operations between the Wolfram Language and MXNet are not reversible. If you pay attention, the network input, exported to MXNet format, was set as a real number, unlike the network input imported in MXNet format, which marks that the input is an array with specifying dimensions.

When constructing a neural network, there is no restriction on how many net chains or net graphs a net can have. For instance, the following example is a neural network from the Wolfram Neural Net Repository, which has a deeper sense of construction (see Figure 9-48). This net is called CapsNet, which is used to estimate the depth map of an image. To consult the net, enter `NetModel["CapsNet Trained`

on MNIST Data,” “DocumentationLink”] for the documentation web page; for the notebook on the Wolfram Cloud, enter NetModel[“CapsNet Trained on MNIST Data,” “ExampleNotebookObject”] or just ExampleNotebook for the desktop version.

```
In[103]:= NetModel["CapsNet Trained on MNIST Data"]
Out[103]=
```

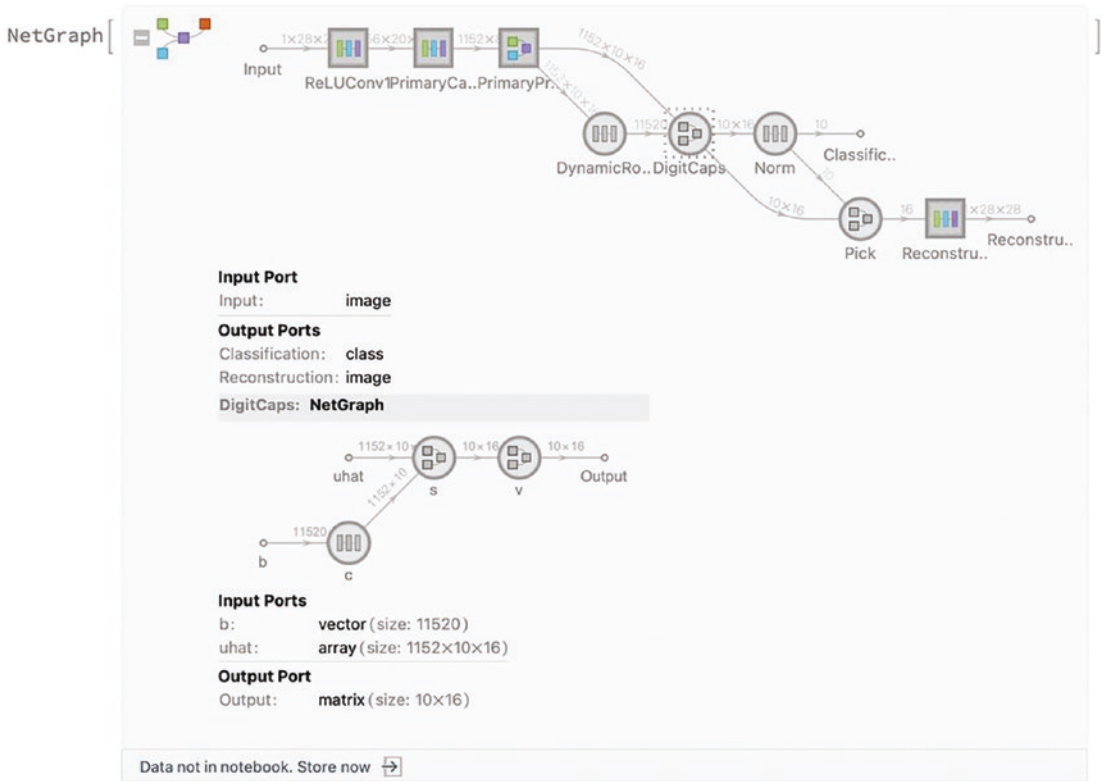


Figure 9-48. CapsNet neural net model

# Summary

This chapter introduced the neural network scheme in the Wolfram Language and covered basic layers components: data input, weight, and biases. Additionally, the chapter focuses on the encoders and decoders, explaining its structure.

## CHAPTER 10

# Neural Networks Framework

This chapter explores training a neural network model in the Wolfram Language, how to access the results and the trained network. You review the basic commands to export and import a net model. You end the chapter by exploring the Wolfram Neural Net Repository and reviewing the LeNet network model.

## Training a Neural Network

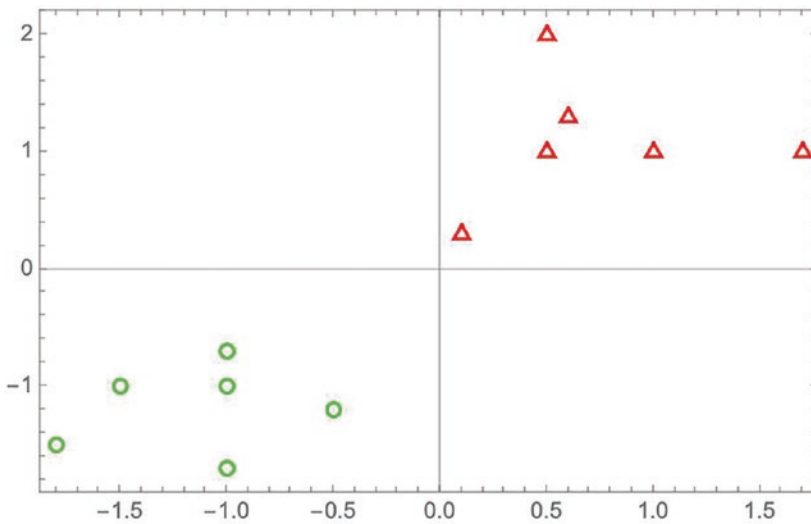
The Wolfram Language contains a very useful command that automates neural network model training. This command is `NetTrain`. Training a neural network consists of fine-tuning the internal parameters of the neural network. The whole point is that the parameters can be learned during training. This general process is done by an optimization algorithm called gradient descent, which is computed with the backpropagation algorithm.

## Data Input

With `NetTrain`, data can be entered in different forms. First, the net model goes as the first argument, followed by the input  $\rightarrow$  target,  $\{\text{inputs}, \dots\} \rightarrow \{\text{target}, \dots\}$  or the name of the data or dataset. Once the net model is defined, the next argument is the data, followed by an optional argument of `All`. The `All` option creates a `NetTrainResultsObject`, which shows the `NetTrain` results panel after the computation and stores all relevant information about the trained model. The options for training the model are entered as the last arguments. Standard options used in layers and containers are available in `NetTrain`.

The next example uses the perceptron model to build a linear classifier. The data to be classified is shown in the following plot (see Figure 10-1).

```
In[1]:= plt=ListPlot[{{{-1.8,-1.5},{-1,-1.7},{-1.5,-1},{-1,-1},{-0.5,-1.2},
{-1,-0.7}}, {{1,1}, {1.7,1}, {0.5,2}, {0.1,0.3}, {0.5,1}, {0.6,1.3}}},
PlotMarkers->"OpenMarkers",Frame->True,PlotStyle->{Green,Red}]
Out[1]=
```



**Figure 10-1.** ListPlot showing two different plot points

Let's define the data, target values, and the training data.

```
In[2]:=data={{-1.8,-1.5},{-1,-1.7},{-1.5,-1},{-1,-1},{-0.5,-1.2},{-1,-0.7},
{1,1},{1.7,1},{0.5,2},{0.1,0.3},{0.5,1},{0.6,1.3}};
target={-1,-1,-1,-1,-1,-1,1,1,1,1,1,1};
trainData=MapThread[#1->]{#2}&,{Standardize[data],target},1];
```

The Standardize function is crucial in the latter code because it normalizes the input data before training the neural network. This step ensures that each feature contributes equally to the learning process during the training phase, preventing any single feature from dominating the others. This process can lead to faster convergence during training and improves the overall performance of the net model. Next, let's define the net model.

```
In[3]:= model=NetChain[{LinearLayer[1,"Input"->2],
ElementwiseLayer[Ramp[#]&]}];
```

## Training Phase

Having prepared the data and the model, you proceeded to train the model. Once the training begins, a progress information panel appears with four main results.

- Summary: contains relevant information about the batches, rounds, and time rates
- Data: involves processed data information
- Method: shows the method used, batch size, and device used for training
- Round: the current state of loss value

```
In[6]:=net=NetTrain[model,trainData,All,LearningRate->0.01,
PerformanceGoal->"TrainingSpeed",TrainingProgressReporting->"Panel",
TargetDevice->"CPU", RandomSeeding->88888,WorkingPrecision->"Real64"]
Out[6]=
```

Figure 10-2 shows the loss plot against the training rounds.

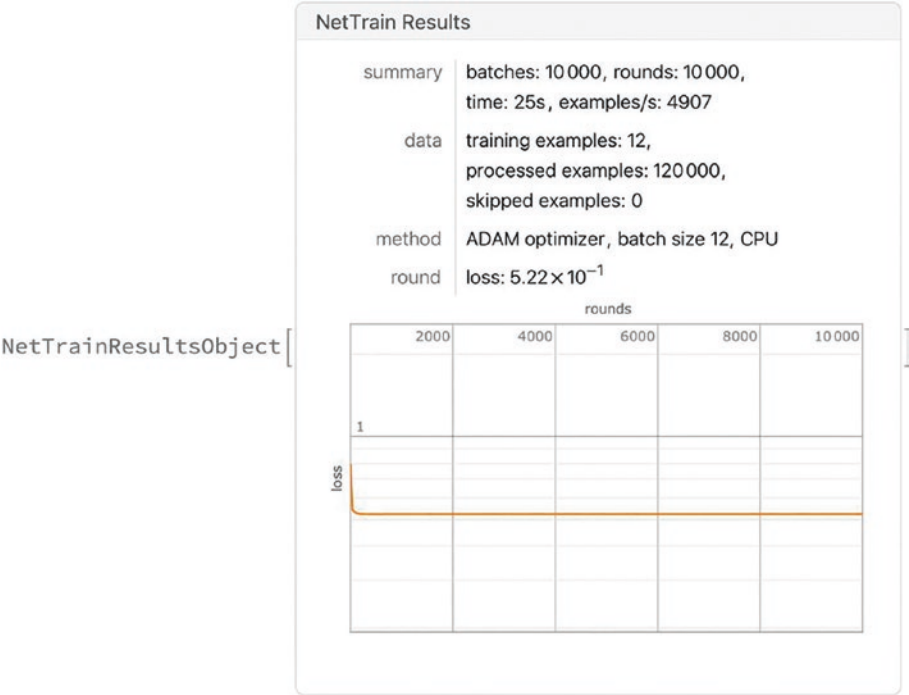


Figure 10-2. NetTrainResultsObject

The Adam optimizer is a variant of the Stochastic gradient descent, which you see later. The object generated is called NetTrainResultsObject.

## Model Implementation

Once the training is done, getting the trained net and model implementation is as follows in Figure 10-3.

```
In[7]:= trainedNet1=net["TrainedNet"]
Out[7]=
```

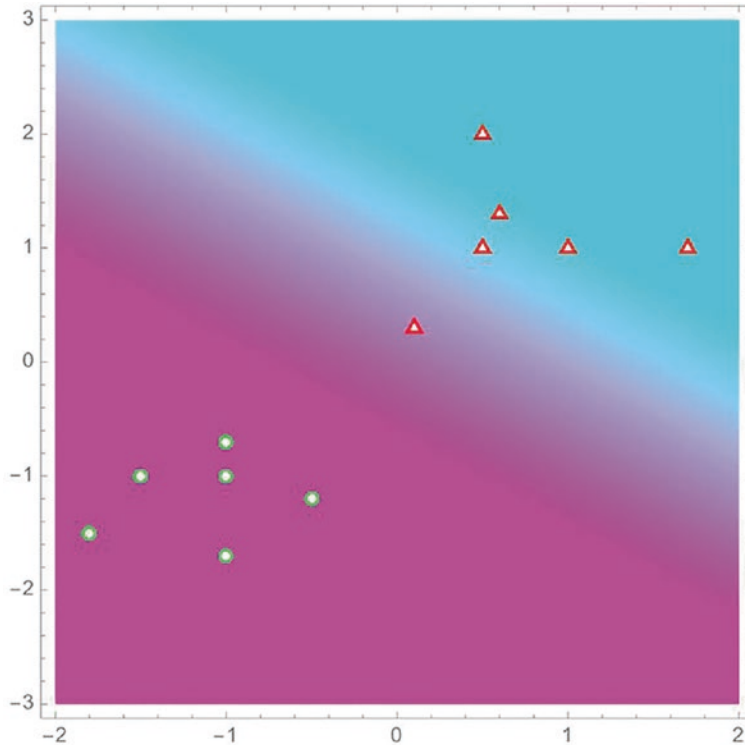


Figure 10-3. Extracted trained net



Let's look at how the trained net identifies each point by plotting the boundaries with a density plot (see Figure 10-4).

```
In[8]:= Show[DensityPlot[trainedNet1[{x,y}],{x,-2,2},{y,-3,3},PlotPoints->
50,ColorFunction->(RGBColor[1-#,2*#,1]&)],Plt]
Out[8]=
```



**Figure 10-4.** Net classification plot

The graphic shows that the boundaries are not well defined and that points near zero might be misclassified. This result can be attributed to the ramp function, which gives 0 if it receives any negative number, but for any positive value, it returns that value. This model can still be improved, perhaps by changing the activation function to a hyperbolic tangent to have robust boundaries.

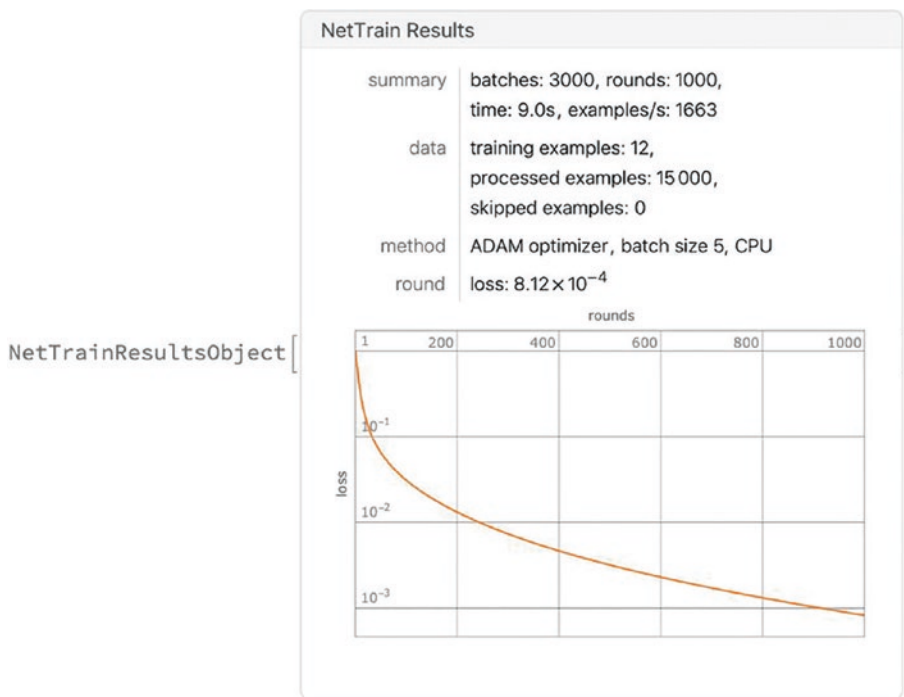
## Batch Size and Rounds

If the batch size is not indicated, it has an automatic value, almost always a value of 64 or powers of two. Remember that the batch size indicates the number of examples the model uses in training before updating the internal parameters of the model. The number of batches is the division of the examples within the training dataset by the batch size. The processed examples are the number of rounds (epochs) multiplied by the number of training examples. The batch size is generally chosen to divide the training set's size evenly. The `MaxTrainingRounds` option determines the number of times the training dataset is passed through during the training phase. When you go through the entire training set just once, it's called an epoch. To better understand this, a batch size of 12 was automatically chosen in the earlier example, which is equal to the number of examples in the training set. This means that it enters a batch of  $12/12 \rightarrow 1$  for epoch or round. Now, the number of epochs was automatically chosen to 10000; this tells you that there are  $1 * 10000$  batches. Also, the number of processed examples is  $12 * (10000)$ , which is equal to 120000. If the batch size does not evenly divide the training set, the final batch has fewer examples than the other batches.

Furthermore, adding a loss function layer to the container or the loss with the `LossFunction -> Loss Layer` option has the same effect. In this case, you use the `MeanSquaredLossLayer` as the loss function option, change the activation function to `Tanh[x]`, set the `Batchsize` to 5, and adjust `MaxTrainingRounds` to 1000.

```
In[9]:= net2=NetTrain[NetChain[{LinearLayer[1,"Input"->2], ElementwiseLayer
[Tanh[#]&]}],trainData,All,LearningRate->0.01, PerformanceGoal->
"TrainingSpeed",TrainingProgressReporting->"Panel", TargetDevice->
"CPU",RandomSeeding->88888,WorkingPrecision->"Real64", LossFunction->
MeanSquaredLossLayer[],BatchSize->5,MaxTrainingRounds->1000]
Out[9]=
```

Figure 10-5 shows that the loss has dropped considerably.

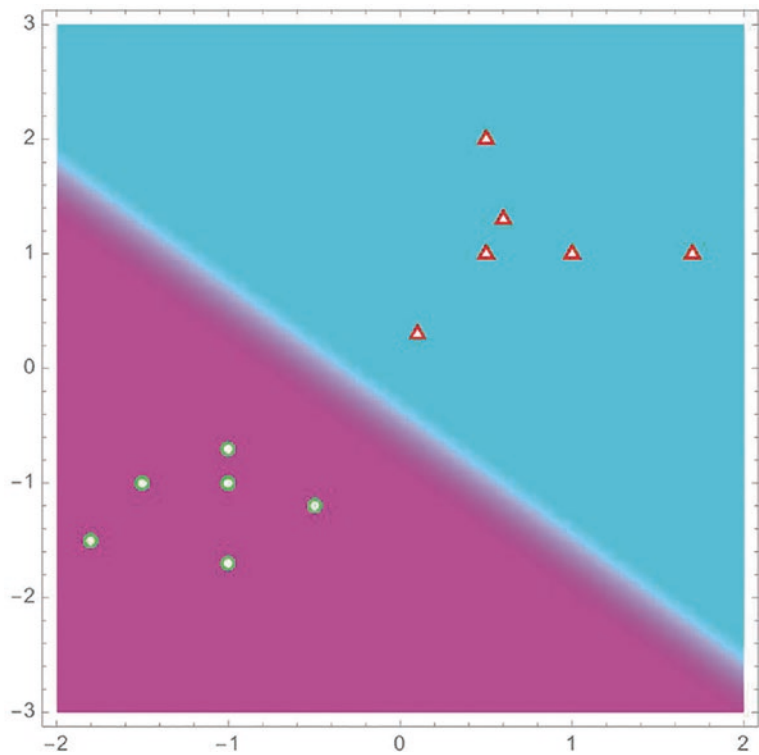


**Figure 10-5.** Training results of the Net2

Let's determine the classification.

```
In[10]:= trainedNet2=net2["TrainedNet"];
Show[DensityPlot[trainedNet2[{x,y}],{x,-2,2},{y,-3,3}, PlotPoints->50,
ColorFunction->(RGBColor[1-#,2*#,1]&)],Plt]
Out[11]=
```

Figure 10-6 shows how the two boundaries are better denoted.



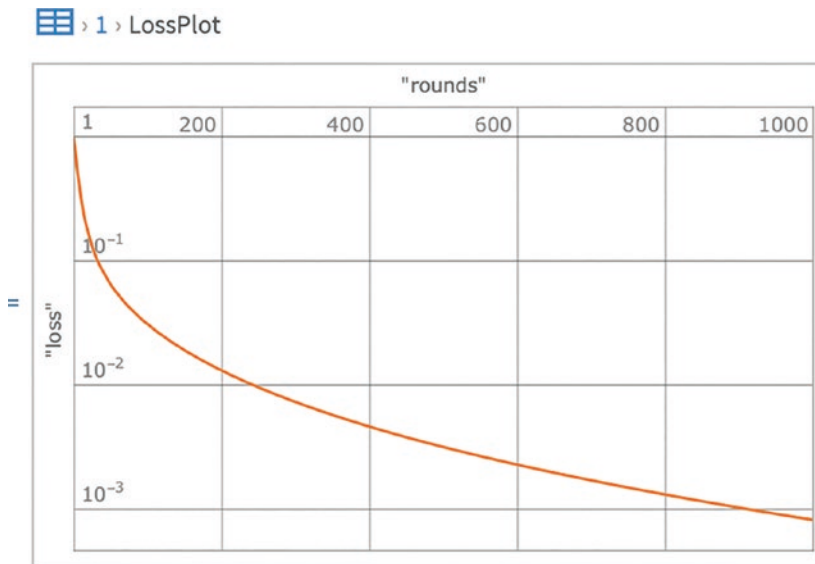
**Figure 10-6.** *Net2 classification plot*

The previous models represent a prediction of a linear layer, in which this classification is compared with the targets so that the error is less and less.

To obtain the graph that shows the value of the error according to the number of rounds carried out in the training, you do it through the properties of the trained network. You can also see the network model's appearance once the loss function is added.

```
In[12]:= Dataset[{Association["LossPlot"->net2["LossPlot"]],
Association["NetGraph"->net2["TrainingNet"]}]}
Out[12]=
```

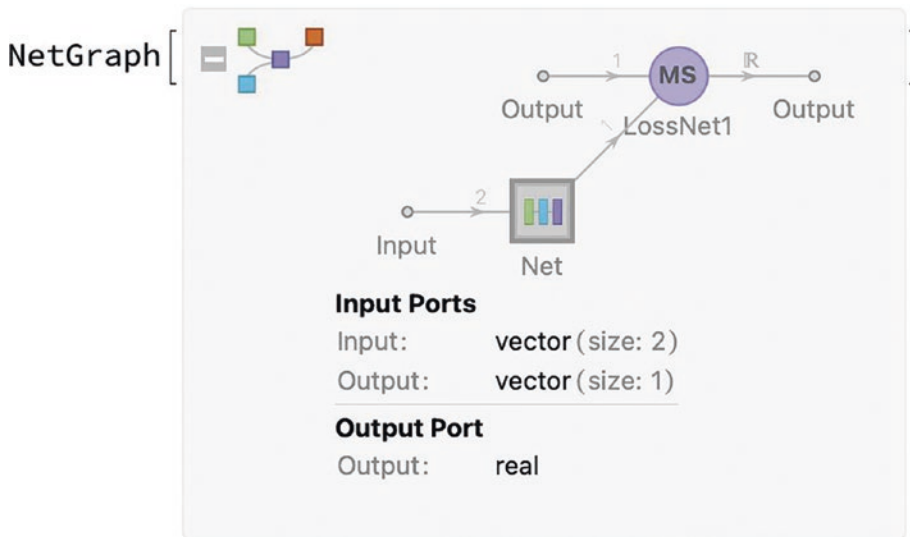
Figure 10-7 shows the loss graph as it decreases rapidly according to the number of rounds.



**Figure 10-7.** *LossPlot contained in the dataset*

To see the network used for training, execute the next code. Mathematica automatically adds a loss function to the neural network (see Figure 10-8) based on the model's layers.

```
In[13]:= net2["TrainingNet"]
Out[13]=
```



**Figure 10-8.** Network model before the training phase

To see the model's properties, you add the string `Properties` as an argument.

```
In[14]:= net2["Properties"]
Out[14]= {ArraysLearningRateMultipliers,BatchesPerRound,BatchesPerSecond,Ba
tchLossList,BatchMeasurements,BatchMeasurementsLists,BatchSize,BestValidati
onRound,CheckpointingFiles,ExamplesProcessed,FinalLearningRate,FinalPlots,I
nitialLearningRate,InternalVersionNumber,LossPlot,MeanBatchesPerSecond,Mean
ExamplesPerSecond,NetTrainInputForm,OptimizationMethod,ReasonTrainingStoppe
d,RoundLoss,RoundLossList,RoundMeasurements,RoundMeasurementsLists,RoundPos
itions,SkippedTrainingData,TargetDevice,TotalBatches,TotalRounds,TotalTrain
ingTime,TrainedNet,TrainingExamples,TrainingNet,TrainingUpdateSchedule,Vali
dationExamples,ValidationLoss,ValidationLossList,ValidationMeasurements,Val
idationMeasurementsLists,ValidationPositions}
```

## Training Method (NetTrain)

Let's look at the training method for the previous network with `OptimizationMethod`. Some variants of the gradient descent algorithm are related to batch size. The first one is the stochastic gradient descent (SGD). The SGD takes a single training batch at a time before taking another step. This algorithm goes through the training examples in a stochastic form—without a sequential pattern and only one instance at a time. The

second variant is the batch gradient descent, meaning that the batch size is set to the size of the training set. This method utilizes all training examples and makes only one update of the internal parameters. The third variant is the mini-batch gradient descent, which consists of dividing the training set into partitions smaller than the whole dataset to update the model's internal parameters to achieve convergence frequently. To see a mathematical of the SGD and mini-batch SGD, visit the article “Efficient Mini-Batch Training for Stochastic Optimization,” by Mu Li, Tong Zhang, Yuqiang Chen, and Alexander J. Smola (2014, August: pp. 661-670; In *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining*).

```
In[15]:= net2["OptimizationMethod"]
Out[15]= {ADAM, Beta1->0.9, Beta2->0.999, Epsilon->1/100000,
GradientClipping->None, L2Regularization->None, LearningRate->0.01,
LearningRateSchedule->None, WeightClipping->None}
```

The method automatically chosen is the Adam optimizer, which uses the SGD method with an adapted learning rate. The other available methods are the RMSProp, SGD, and the SignSGD. Within the available methods, there are also options to indicate the learning rate, when to scale, when to use the L2 regularization, the gradient, and weight clipping.

## Measuring Performance

In addition to the methods, you can establish what measures to consider during the training phase. These options depend on the type of loss function used and which is intrinsically related to the task, like classification, regression, and clustering. In the case of MeanSquaredLossLayer or MeanAbsoluteLossLayer, the common option is MeanDeviation, which is the absolute value of the average of the residuals. MeanSquare is the mean square of the residuals, RSquared is the coefficient of determination, and standard deviation is the root mean square of the residuals. After completing the training, the measure appears in the net results (see Figure 10-9). The soft sign activation function is used in this example to try out a different activation function and observe its use.

```
In[16]:= net3 = NetTrain[ NetChain[{LinearLayer[1, "Input" -> 2],
ElementwiseLayer["SoftSign"]}], trainData, All, LearningRate -> 0.01,
PerformanceGoal -> "TrainingSpeed", TrainingProgressReporting -> "Panel",
```

```
TargetDevice -> "CPU", RandomSeeding -> 88888, WorkingPrecision ->
"Real64", Method -> "ADAM", LossFunction -> MeanSquaredLossLayer[],
BatchSize -> 5, MaxTrainingRounds -> 1000, TrainingProgressMeasurements ->
{"MeanDeviation", "MeanSquare", "RSquared", "StandardDeviation"}]
Out[16]=
```

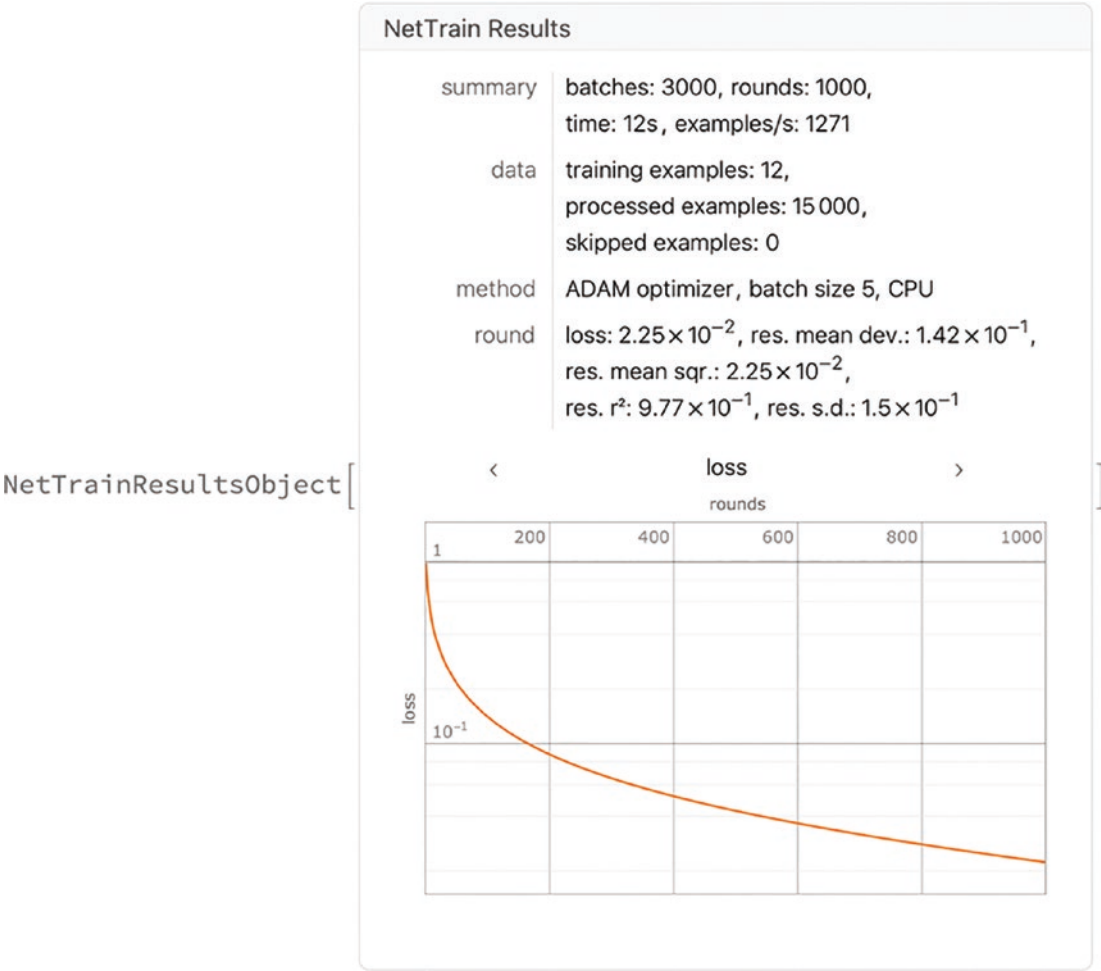


Figure 10-9. Net results with new measures added



## Model Assessment

To access the values of the measures chosen, use the `NetResultsObject`. In the case of the training set values, these are found in the properties of `RoundLoss` (gives the average value of the loss), `RoundLossList` (returns the average values of the loss during training), `RoundMeasurements` (the measurements of the training of the last round), and `RoundMeasurementsLists` (the specified measurements for each round). This result is depicted in Figure 10-10.

```
In[17]:= net3[#]&/{ "RoundMeasurements" }//Dataset[#]&
Out[17]=
```

Loss	0.0224971
MeanDeviation	0.141845
MeanSquare	0.0224971
RSquared	0.977402
StandardDeviation	0.14999

**Figure 10-10.** Dataset with the new measures

To get all the plots, use the `FinalPlots` option.

```
In[18]:= net3["FinalPlots"]//Dataset;
```

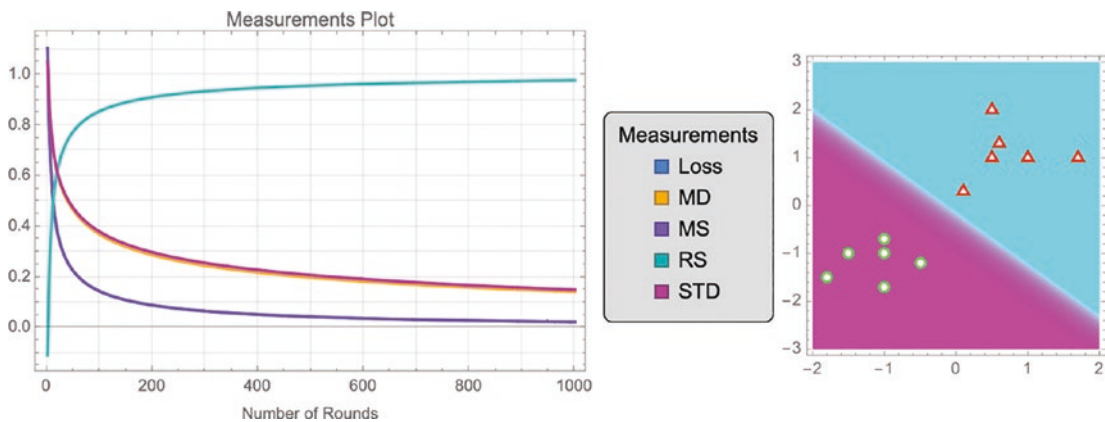
To replicate the plots of the measurements, extract the values of the measurements of each round with `RoundMeasurementsLists`.

```
In[19]:= measures=net3[#]&/{ "RoundMeasurementsLists" };
Keys[measures]
Out[20]= {{Loss,MeanDeviation,MeanSquare,RSquared,StandardDeviation}}
```

Let's plot the values for each round, starting with `Loss` and finishing with `StandardDeviation`. You can also see how the network model makes the classification boundaries (see Figure 10-11).

```
In[21]:= trainedNet3 =
  net3["TrainedNet"]; Grid[{{ListLinePlot[{measures[[1, 1]]
(*Loss*), measures[[1, 2]] (*MeanDeviation*), measures[[1, 3]]
(*MeanSquare*), measures[[1, 4]] (*RSquared*), measures[[1, 5]]
```

```
(*StandardDeviation*)), PlotStyle -> Table[ColorData[101, i], {i,
1, 5}], Frame -> True, FrameLabel -> {"Number of Rounds", None},
PlotLabel -> "Measurements Plot", GridLines -> All, PlotLegends ->
SwatchLegend[{Style["Loss", #], Style["MD", #], Style["MS", #],
Style["RS", #], Style["STD", #]}, LegendLabel -> Style["Measurements", #],
LegendFunction -> (Framed[#, RoundingRadius -> 5, Background -> LightGray]
&)], ImageSize -> Medium] &[Black], Show[DensityPlot[trainedNet3[{x, y}],
{x, -2, 2}, {y, -3, 3}, PlotPoints -> 50, ColorFunction -> (RGBColor[1 - #,
2*#, 1] &)], plt, ImageSize -> 200]]}]
Out[21]=
```



**Figure 10-11.** Round measures plot and density plot

The Loss and MeanSquared have the same values (since the loss is a mean squared error loss function), which is why the two graphics overlap. The mean deviation and standard deviation have similar values but not the same. Three models are constructed, and the activation function changes in each process. Looking at the plots, you see how each function changes how the neural network model learns from the training data. In the previous examples, the graphics were the loss plot for the training process and other measurements related to the means squared loss layer. Make sure to consult the documentation to confirm the measurements' names; remember that not all measurements apply to all loss functions.

In the subsequent section, you see how to generate the loss plot and the validation plot during the training phase to validate that the LeNet model is learning during training and how well the model can perform in data never seen before (validation set).

## Exporting a Neural Network

Once a net model has been trained, you can export this trained net to a WNet format so that the net can be used without the need for training in the future. The export method also works for uninitialized network architectures.

```
In[22]:= Export["/Users/macosex/Desktop/TrainedNet3.
wnet",net3["TrainedNet"]]
Out[22]= /Users/macosex/Desktop/TrainedNet3.wnet
```

Importing them back is done precisely as any other file, but imported elements can be specified. Net imports the net model and all initialized arrays; UninitializedNet and ArrayList imports for the numeric array's objects of the linear layers; ArrayAssociation imports for the numeric arrays in association form, and WVersion is used to see the version of the Wolfram Language used to build the net. The following dataset shows all the options (see Figure 10-12).

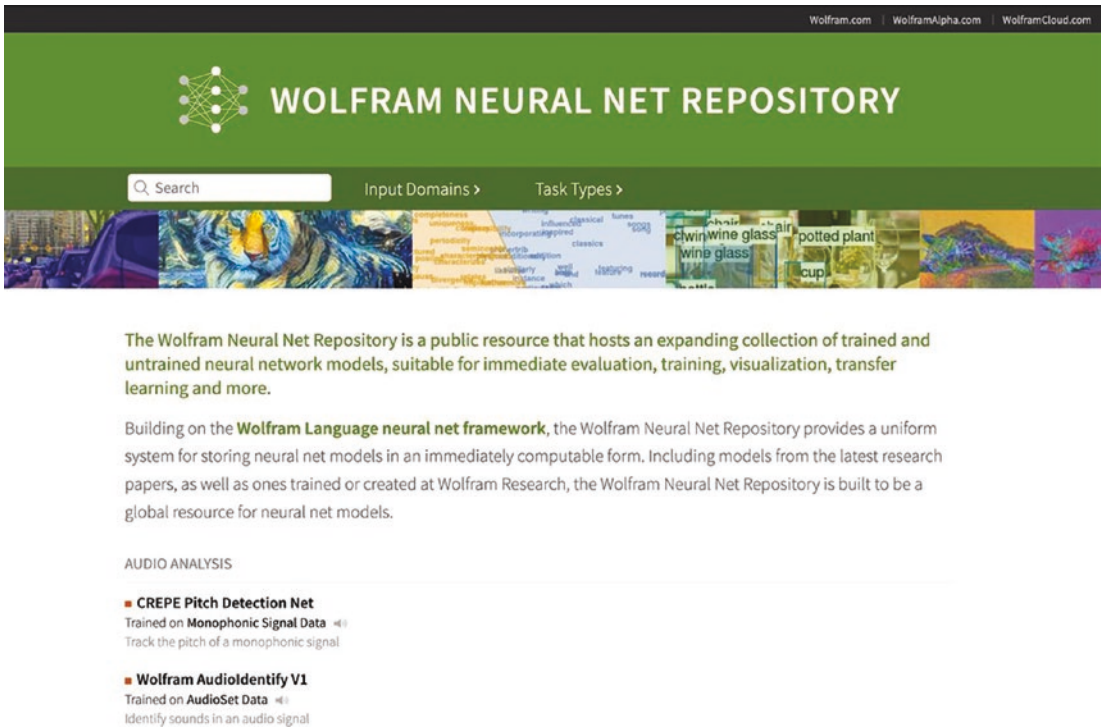
```
In[23]:=Dataset@AssociationMap[Import["/Users/macosex/Desktop/TrainedNet3.
wnet",#]&,{"Net","UninitializedNet","ArrayList","ArrayAssociation",
"WVersion"}]
Out[23]=
```

Net	NetChain[ Input port: vector(size: 2) Output port: vector(size: 1) ]
UninitializedNet	NetChain[ Input port: vector(size: 2) Output port: vector(size: 1) ]
ArrayList	{NumericArray[ Type: Real64 Dimensions: {1, 2} ], NumericArray[ Type: Real64 Dimensions: {1} ]}
ArrayAssociation	< {1, Weights} → NumericArray[ Type: Real64 Dimensions: {1, 2} ], {1, Biases} → NumericArray[ Type: Real64 Dimensions: {1} ] >
WVersion	13.3.0

**Figure 10-12.** Dataset with the available import options

# Wolfram Neural Net Repository

The Wolfram Neural Net Repository is a free-access website containing a repertoire of various pre-trained neural network models. The models are categorized by the input and data types, be it audio, image, numeric array, or text. Furthermore, they are also categorized by the kind of task they perform, from audio analysis or regression to classification. The main page of the website is shown in Figure 10-13.

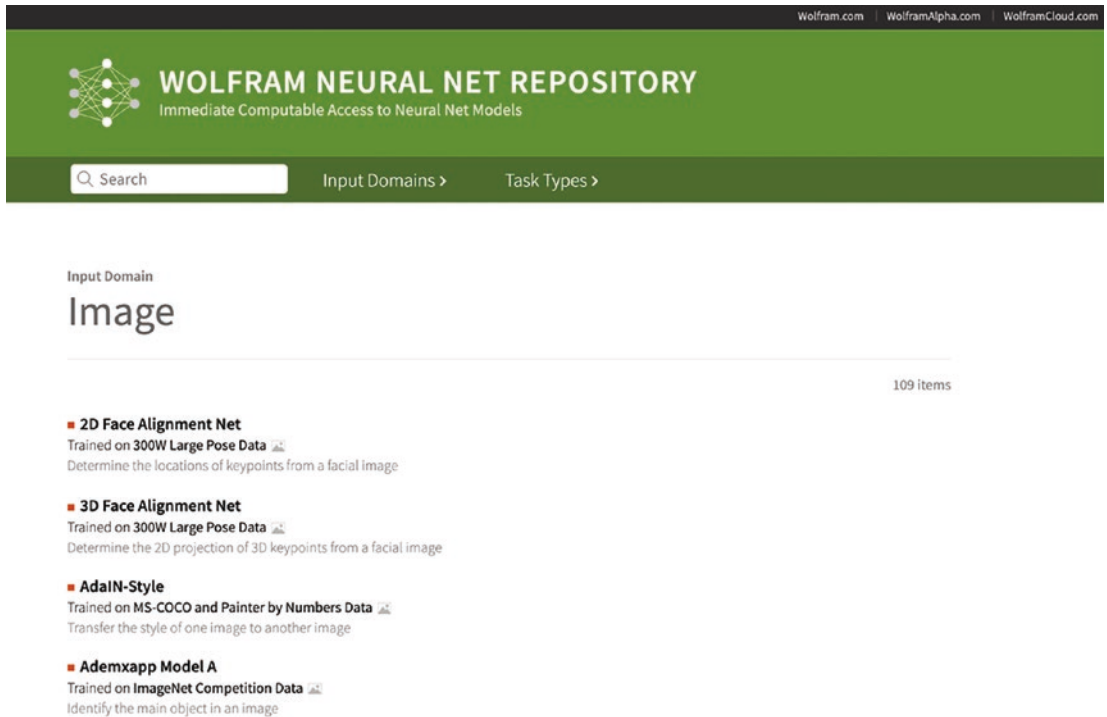


**Figure 10-13.** *Wolfram Neural Net Repository home page*

Enter <https://resources.wolframcloud.com/NeuralNetRepository/> in your favorite browser to access the web page, or run `SystemOpen` from Mathematica, which opens the web page in the system's default browser.

Once the site is loaded, net models can be browsed by either input or task. The models in this repository are built in the Wolfram Language, allowing you to use them within Mathematica. This leads to the models being found in a form that can be accessed from Mathematica or the Wolfram Cloud for prompt execution. If you scroll down, you see that the models are structured by name and the data used for training, along with a short description. Such is the case, for example, for the Wolfram AudioIdentify V1

network, which is trained with the AudioSet Data and identifies sounds in audio signals. To browse categories, you can choose the category from the menu. Figure 10-14 shows the site's appearance after an input category is chosen; in this case, the neural networks that receive images as inputs.



*Figure 10-14. Category site, based on the input image*

## Selecting a Neural Net Model

Once a category is chosen, it shows all the net models associated with the selected input category. Like with the Wolfram Data Repository, once the model is selected, it shows relevant information, like in Figure 10-15, where the selected net model is the neural network Wolfram ImageIdentify Net V1.



# WOLFRAM NEURAL NET REPOSITORY

Immediate Computable Access to Neural Net Models

[Input Domains >](#)

[Task Types >](#)

## Wolfram ImageIdentify Net V1

Identify the main object in an image

Released in 2017 by Wolfram Research, this net was trained on over 4,000 classes of objects. It is part of the back end for the ImageIdentify function in Wolfram Language 11.1. It was designed to achieve a good balance among classification accuracy, size and evaluation speed.

Number of layers: 232 | Parameter count: 14,713,147 | Trained size: 65 MB |

### TRAINING SET INFORMATION

Internal Wolfram ImageIdentify training set, consisting of over 3 million training images and over 4,000 classes of objects (not publicly available).

Examples

Download Example Notebook

Open in Wolfram Cloud

> Resource retrieval

> Basic usage

> Feature extraction

> Visualize convolutional weights

> Transfer learning

> Net information

> Export to MXNet

**Figure 10-15.** Wolfram ImageIdentify Net V1

It is possible to navigate from the website and download the notebook containing the network model, but it is also possible from Mathematica. In other words, search for network models through ResourceSearch. The example shows the search if you were interested in knowing the models of the networks that contain the word image (see Figure 10-16).

```
In[24]:= ResourceSearch[{"Name"->"Image", "ResourceType"-> "NeuralNet"}]
//Dataset[#,MaxItems->{4,3}]&
Out[24]=
```



Name	ResourceType	ResourceObject
Colorful Image Colorization Trained on ImageNet Competition Data	NeuralNet	ResourceObject["Colorful image Colorization Trained on imageNet Competition Data"]
ColorNet Image Colorization Trained on ImageNet Competition Data	NeuralNet	ResourceObject["ColorNet Image Colorization Trained on imageNet Competition Data"]
EfficientNet Trained on ImageNet	NeuralNet	ResourceObject["EfficientNet Trained on ImageNet"]
Wolfram ImageIdentify Net V1	NeuralNet	ResourceObject["Wolfram ImageIdentify Net V1"]

**Figure 10-16.** Resource Dataset

The dataset shown in Figure 10-16 has only three columns for display purposes, but you can navigate through the entire dataset using the slider. The columns not shown in the image are Description, Location, and DocumentationLink. The last column provides the link that leads to the web model page.

## Accessing Inside Mathematica

To access the model architecture, add the object argument; for example, do the following for the Wolfram ImageIdentify Net V1 Network (see Figure 10-17).

```
In[25]:= ResourceSearch[{"Name"->"Wolfram ImageIdentify", "ResourceType"->
"NeuralNet"}, "Object"]
Out[25]=
```

```
{ResourceObject[
  {
    Name: Wolfram ImageIdentify Net V1 »
    Type: NeuralNet
    Description: Identify the main object in an image
    ByteCount:
    TrainingSetInformation:
    InputDomains: Image
    TaskType: Classification
    Keywords: ImageIdentify, object classification
    Data Location: None
    UUID: 044dd5d5-5895-4252-889f-f67729b1a6d3
    Version: 1.10.0
    Elements: ConstructionNotebook, EvaluationNet, UninitializedEvaluationNet, ConstructionNotebookExpression, EvaluationExample
  }
]}
```

**Figure 10-17.** Wolfram ImageIdentify Net V1 resource

---

**Note** To avoid problems accessing the Wolfram Net Repository from Mathematica, ensure you are logged in to the Wolfram Cloud or your Wolfram account.

---



The following code is suppressed here to access the pre-trained model, but removing the semicolon returns the NetChain object of the pre-trained neural network.

```
In[26]:= ResourceSearch[{"Name"->"Wolfram ImageIdentify", "ResourceType"->
"NeuralNet"}, "Object"][[1]]//ResourceData;
Out[26]=
```

## Retrieving Relevant Information

Information about the model is accessed from ResourceObject. The following is the relevant information from the ImageIdentify model in a dataset (see Figure 10-18). To see all information in the dataset format, type ResourceObject["Wolfram ImageIdentify Net V1"][All]//Dataset [#] &.

```
In[27]:= Dataset[AssociationMap[ResourceObject["Wolfram ImageIdentify Net V1"],
{"Name", "RepositoryLocation", "ResourceType", "ContentElements", "Version",
"Description", "TrainingSetInformation", "InputDomains", "TaskType", "Keywords",
"Attributes", "LatestUpdate", "DownloadedVersion", "Format",
"ContributorInformation", "DOI", "Originator", "ReleaseDate", "ShortName",
"WolframLanguageVersionRequired"}]]
Out[27]=
```

Name	Wolfram ImageIdentify Net V1
RepositoryLocation	<a href="https://www.wolframcloud.com/obj/resourcesystem/ap...">https://www.wolframcloud.com/obj/resourcesystem/ap...</a>
ResourceType	NeuralNet
ContentElements	{ConstructionNotebook, EvaluationNet, UninitializedEvaluationNet, ConstructionNotebookExpression, EvaluationExample}
Version	1.10.0
Description	Identify the main object in an image
TrainingSetInformation	Internal Wolfram ImageIdentify training set, consisting of over 3 million training images and over 4,000 classes of objects (not publicly available).
InputDomains	Image
TaskType	Classification
Keywords	{ImageIdentify, object classification}
Attributes	{LocalCopyable, CloudCopyable, Multipart}
LatestUpdate	Fri 28 Feb 2020 00:00:00
DownloadedVersion	1.10.0
Format	<  EvaluationNet → WLNet, UninitializedEvaluationNet → WLNet, ConstructionNotebookExpression → NB, EvaluationExample → WXF  >
ContributorInformation	<  PublisherID → Wolfram, DisplayName → Wolfram Research  >
DOI	<a href="https://doi.org/10.24097/wolfram.34204.data">https://doi.org/10.24097/wolfram.34204.data</a>
Originator	Wolfram Research
ReleaseDate	Mon 20 Feb 2017 16:00:00
ShortName	Wolfram-ImageIdentify-Net-V1
WolframLanguageVersionRequired	11.1

**Figure 10-18.** Dataset of some properties of the Wolfram ImageIdentify Net V1



Here, in a few steps, is the way to access the trained neural network and much relevant information associated with the neural network. It should be noted that the process is also used to find other resources in the Wolfram Cloud or local resources, not only neural networks, since, in general, ResourceSearch looks for an object within the Wolfram Resource System. Such is the case of the neural network models in the Wolfram Neural Net Repository.

## LeNet Neural Network

The following example examines a neural network model named LeNet. Despite being able to access the model from a Wolfram resource, as you saw previously, performing operations with networks found in the Wolfram Neural Net Repository with the NetModel command is possible. To get a better idea of how this network is used, let's first look at the description of the network, its name, how it is used, and where it was proposed for the first time.

### LeNet Model

The neural network LeNet is a convolutional neuronal network within the deep learning field. The neural network LeNet is recognized as one of the first convolutional networks that promoted deep learning. This network was used for character recognition to identify handwritten digits. Today, architectures are based on LeNet neural network architecture, but you focus on the Wolfram Neural Net Repository version. This architecture consists of four key operations: convolution, non-linearity, subsampling, or pooling and classification. To learn more about the LeNet convolutional neural network, see *Neural Networks and Deep Learning: A Textbook* by Charu C. Aggarwal (Springer, 2018). With NetModel, you can obtain information about the LeNet network that has been previously trained.

```
In[28]:= NetModel["LeNet Trained on MNIST Data",#]&/@{"Details","ShortName",
,"TaskType","SourceMetadata"}//Column
Out[28]= This pioneer work for image classification with convolutional
neural nets was released in 1998. It was developed by Yann LeCun and his
collaborators at AT&T Labs while they experimented with a large range of
machine learning solutions for classification on the MNIST dataset.
```

LeNet-Trained-on-MNIST-Data

```
{Classification} <|Citation->Y. LeCun, L. Bottou, Y. Bengio, P. Haffner,
"Gradient-Based Learning Applied to Document Recognition," Proceedings of
the IEEE, 86(11), 2278-2324 (1998),Source->http://yann.lecun.com/exdb/
lenet,Date->DateObject[{1998},Year,Gregorian,-5.]]>
```

---

**Note** To access all the properties of a model with `NetModel`, add properties as the second argument—`NetModel["LeNet Trained on MNIST Data," "Properties"]`.

---

The input this model receives consists of images in grayscale with a size of 28 x 28, and the model's performance is 98.5% on the MNIST dataset.

```
In[29]:= NetModel["LeNet Trained on MNIST Data",#]&@{"TrainingSetInformati
on","InputDomains","Performance"}//Column
Out[29]= MNIST Database of Handwritten Digits, consisting of 60,000
training and 10,000 test grayscale images of size 28x28.
{Image}
This model achieves 98.5% accuracy on the MNIST dataset.
```

## MNIST Dataset

This network is used for rating, just as it appears in `TaskType`. The digits are in a database known as the MNIST database. The MNIST database is an extensive database of handwritten digits (see Figure 10-19) that contains 60,000 images for training and 10,000 for testing, the latter being used to get a final estimate of how well the neural net model works. To observe the complete dataset, you load it from the Wolfram Data Repository with `ResourceData` and `ImageDimensions` to verify that the dimensions of the pictures are 28 x 28 pixels.

```
In[30]:= (*This is for seven elements randomly sampled, but you can check
the whole data set.*)
TableForm[
  SeedRandom[900];
  RandomSample[ResourceData["MNIST", "TrainingData"], 7],
  TableDirections -> Row]
```

```
Map[ImageDimensions, %[[1 ;; 7, 1]]]
(*Test set : ResourceData["MNIST","TestData"] *)
Out[30]//TableForm=
```

**Figure 10-19.** A random sample of the MNIST training set

```
Out[31]= {{28,28},{28,28},{28,28},{28,28},{28,28},{28,28},{28,28}}
```

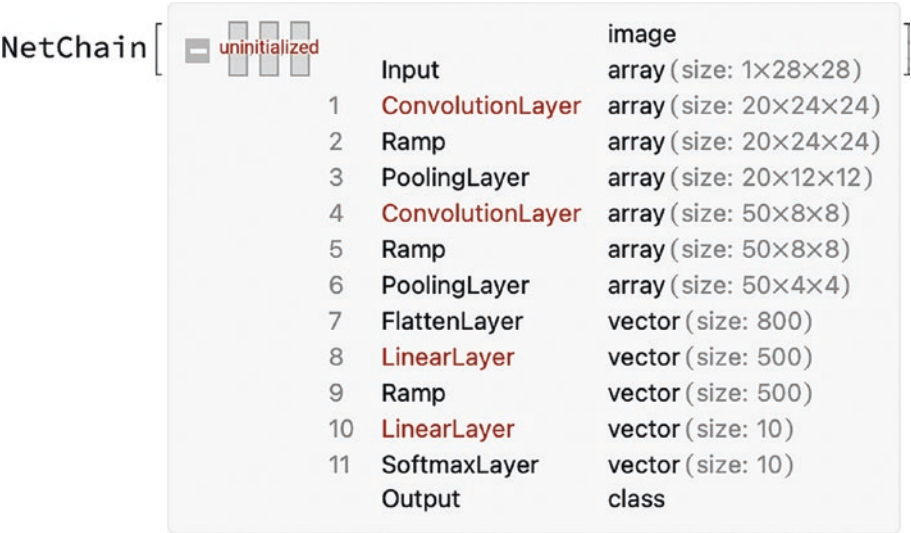
Figure 10-19 shows the images of the digits, the class to which they apply, and the dimensions of each image. You extract the training sets and test sets, which you use later.

```
In[32]:= {trainData,testData}={ResourceData["MNIST","TrainingData"],
ResourceData["MNIST","TestData"] };
```

## LeNet Architecture

Let's start by downloading the neural network from the NetModel command, which extracts the model from the Wolfram Neural Net Repository. The next exercise loads the network that has not been trained since you do the training and validation process. It should be noted that the LeNet model in the Wolfram Language is a variation of the original architecture (see Figure 10-20).

```
In[33]:= uninitLeNet=NetModel["LeNet Trained on MNIST Data",
"UninitializedEvaluationNet"](*To work locally with the untrained
model: NetModel["LeNet"]*)
Out[33]=
```



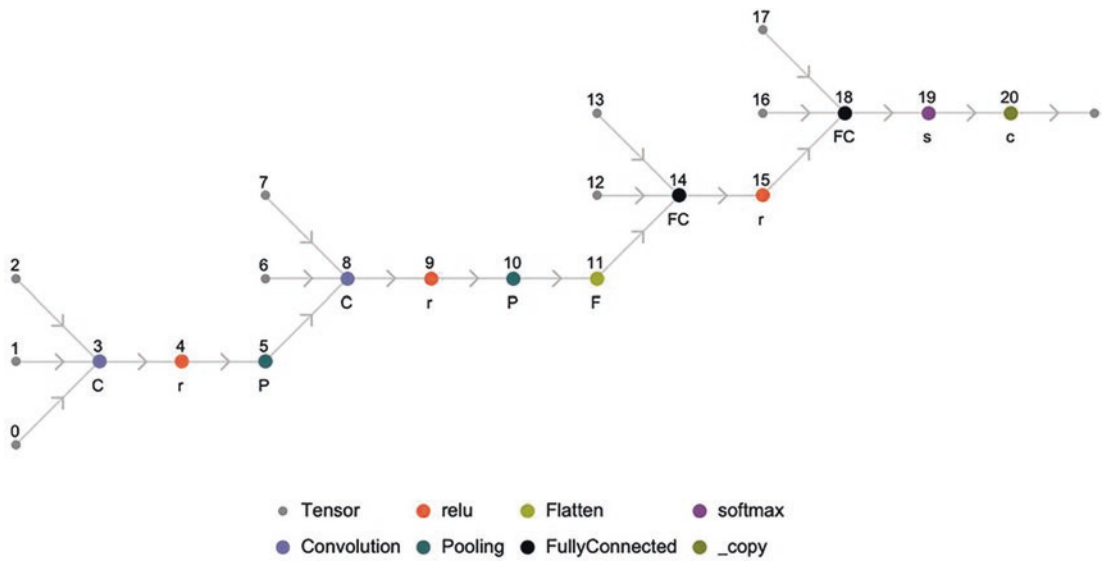
**Figure 10-20.** *LeNet architecture*

The LeNet network in the Wolfram Neural Net Repository is built from 11 layers. The layers that appear in red are layers with learnable parameters: two convolutional layers and two linear layers.

## MXNet Framework

With the MXNet framework, let's first visualize the process of this network through the MXNet operation graph (see Figure 10-21).

```
In[34]:= Information[uninitLeNet,"MXNetNodeGraphPlot"]
Out[34]=
```



**Figure 10-21.** MXNet graph of the LeNet architecture

LeNet architecture starts at the input with the operation that converts the image to a numeric array, followed by the first operation. This convolution returns a 20-feature map with a rectified linear unit (ReLU) activation function immediately following nodes 3 and 4. Then, the first max-pooling operation (subsampling layers) selects the maximum value in the pooling node 5. Then, the second convolutional operation returns a 50-feature map with a ReLU activation function immediately following nodes 8 and 9. The last convolution operation is followed by another max-pooling operation (node 10), followed by a flattening operation (node 11), which flattens the output of the pooling operation into a single vector. The last pooling operation gives an array of  $50 \times 4 \times 4$ , and the flatten operation returns an 800-vector that is the input of the next operation. Next, you see the first fully connected layer (node 14); the first fully connected layer has a ReLU function (node 15), and the second fully connected layer has the softmax function (node 19). The last fully connected layer can be interpreted as a multilayer perceptron (MLP) that normalizes the output into a probability distribution to indicate the probability of each class. Finally, the tensor is converted to a class with the decoder. Nodes 4, 9, and 15 are the layers for non-linear operations (ReLU), and node 19 applies the softmax function for output classification. In summary, the architecture is as follows: Tensor (input), Convolution, ReLU, Pooling, Convolution, ReLU, Pooling, Flatten, Fully Connected (with ReLU), Fully Connected (with softmax), and Class output.

## Preparing LeNet

Since LeNet is a neural network for image classification, an encoder and decoder must be used. The NetEncoder is inserted in the input NetPort, and the NetDecoder is on the output NetPort. Looking into the NetGraph (see Figure 10-22) might be useful in understanding the process inside the Wolfram Language. Clicking the input and output shows the relevant information.

```
In[35]:= NetGraph[uninitLeNet]
Out[35]=
```



**Figure 10-22.** NetGraph of the LeNet model

You can extract the encoder and decoder to inspect their infrastructure. The encoder receives an image of the dimensions of 28 x 28 of any color space and encodes the image into a color space set to grayscale, returning then an array of the size of 1 x 28 x 28. On the other hand, the decoder is a class decoder that receives a 10-size vector, which tells the probability for the class labels that are 0, 1, 2, 3, 4, 5, 6, 7, 8, and 9.

```
In[36]:={enc=NetExtract[uninitLeNet,"Input"],dec=NetExtract[uninitLeNet,
"Output"]};//Row;
```

First, let's look at how the net model works with NetInitialize; for example, use an image of 0 in the training set.

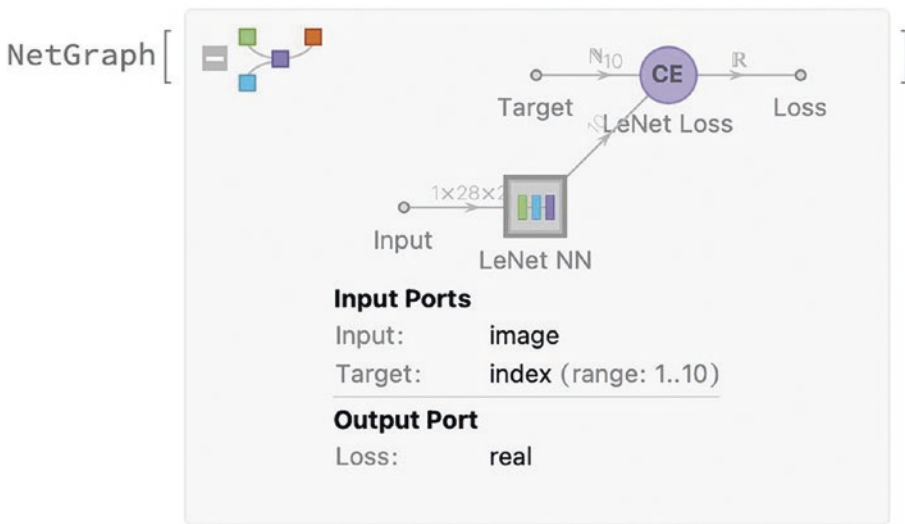
```
In[37]:= testNet=NetInitialize[uninitLeNet,RandomSeeding->8888];
testNet@trainData[[1,1]](*TrainData[[1,1]] belongs to a zero*)
Out[38]= 9
```

The net returns that the image belongs to class 9, which means that the image is a number 9; clearly, this is wrong. Let's try NetInitialize again but with the different methods available. Writing all, as the second argument to NetInitialize, overwrites any pre-existing learning parameters on the network.

```
In[39]:= {net1, net2, net3, net4} = Table[NetInitialize[uninitLe
Net, All, Method -> i, RandomSeeding -> 8888], {i, {"Kaiming",
"Xavier", "Orthogonal", "Identity"}}]; {net1[trainData[[1, 1]]],
net2[trainData[[1, 1]]], net3[trainData[[1, 1]]], net4[trainData[[1, 1]]]}
Out[40]= {9,9,7,3}
```

Every net model fails to classify the image in the correct class. This result is because the neural network has not been trained, unlike `NetInitialize`, which only randomly initializes the learnable parameters without proper training. This is why, with `NetInitialize`, the model fails to classify the image given correctly. But first, let's establish the network graph to better illustrate the model, as seen in Figure 10-23.

```
In[41]:= leNet=NetInitialize[NetGraph[<|"LeNet NN" -> uninitLeNet, "LeNet
Loss" -> CrossEntropyLossLayer@"Index"|>, {NetPort@"Input" -> "LeNet NN",
"LeNet NN" -> NetPort@{"LeNet Loss", "Input"}, NetPort@"Target" -> NetPort@
{"LeNet Loss", "Target"}}], RandomSeeding -> 8888]
Out[41]=
```



**Figure 10-23.** LeNet ready graph

Before you train the net, you must make the validation set suited for the `CrossEntropyLossLayer` in the target input because the classes start at 0 and end at 9, and the Index target begins at 1 and goes on. So, the target input needs to be between 1 and 10.

```
In[42]:= trainDts=Dataset@Join[AssociationThread["Input"->#]& /@Keys[trainData],AssociationThread["Target"-> #]&/@Values[trainData]+1,2];
testDts=Dataset@Join[AssociationThread["Input"->#]& /@Keys[testData],
AssociationThread["Target"-> #]&/@Values[testData]+1,2];
```

The training set and validation set have the form of a dataset. Only four random samples are shown in Figure 10-24.

```
In[44]:= BlockRandom[SeedRandom[999];
{RandomSample[trainDts[[All]],4],RandomSample[testDts[[All]],4}]]
Out[44]=
```

Input	Target	Input	Target
1	2	7	8
9	10	2	3
8	9	4	5
4	5	4	5

**Figure 10-24.** The dataset of the training and test set

## LeNet Training

Now that you have grasped the process of this neural net model, you can proceed to train the neural net model. With `NetTrain`, you gradually modify the learnable parameters of the neural network to reduce the loss. The next training code is set with the options seen in the previous section, but here, you add new options also available for training. The first one is `TrainingProgressMeasurements`. `TrainingProgressMeasurements` can specify measures such as accuracy and precision. These are measured during the training phase

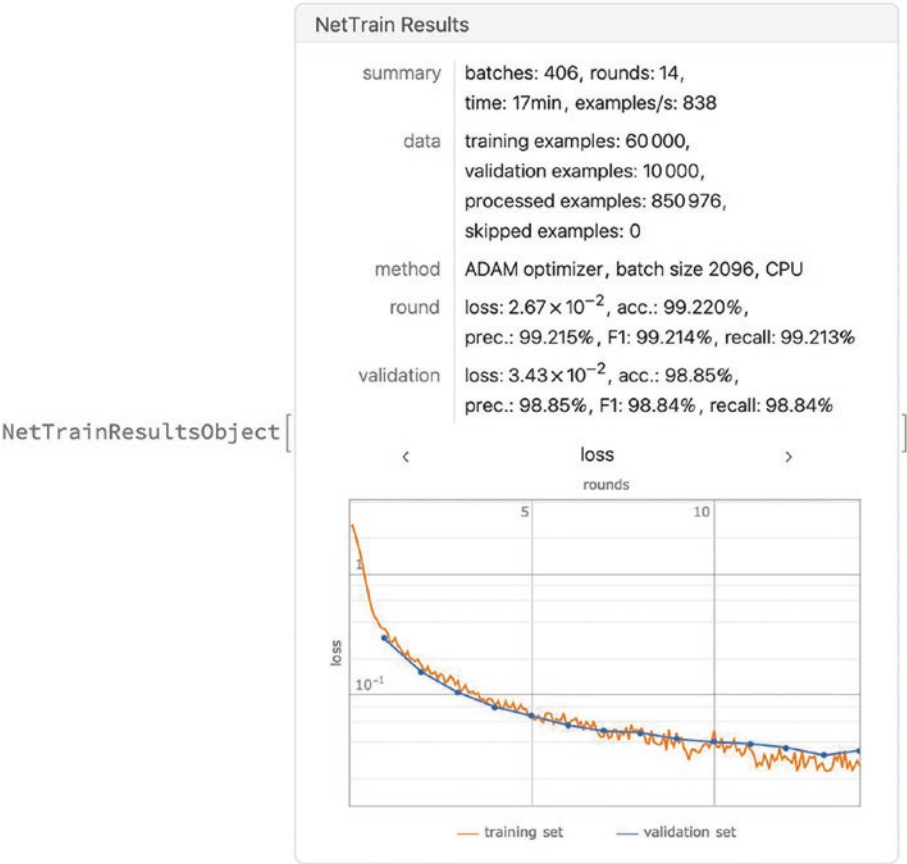


by round or batch. The `ClassAveraging` is used to specify to get the macro-average or the micro-average of the measurement specified `<|"Measurement" -> "measurement"` (Accuracy, RSquared, Recall, MeanSquared, etc.), `"ClassAveraging" -> "Macro"|>`.

The second option is the `TrainingStoppingCriterion`, which is used to add an early stopping to avoid overfitting during the training phase based on different criteria, such as stopping the training when the validation loss is not improving, measuring the absolute or relative change of a measurement (accuracy, precision, loss, etc.), or stopping the training when the loss or other criteria does not improve after a certain number of rounds `<|Criterion->"measurement" (Accuracy, Loss, Recall, etc.), "Patience"-># of rounds|>`.

```
In[45]:= netResults = NetTrain[leNet, trainDts, All, ValidationSet ->
testDts, MaxTrainingRounds -> 15, BatchSize -> 2096, LearningRate ->
Automatic, Method -> "ADAM", TargetDevice -> "CPU", PerformanceGoal
-> "TrainingMemory", WorkingPrecision -> "Real32", RandomSeeding
-> 99999, TrainingProgressMeasurements -> {<|"Measurement" ->
"Accuracy", "ClassAveraging" -> "Macro"|>, <|"Measurement"
-> "Precision", "ClassAveraging" -> "Macro"|>, <|"Measurement"
-> "F1Score", "ClassAveraging" -> "Macro"|>, <|"Measurement"
-> "Recall", "ClassAveraging" -> "Macro"|>, <|"Measurement" ->
"ROCCurvePlot", "ClassAveraging" -> "Macro"|>, <|"Measurement"
-> "ConfusionMatrixPlot", "ClassAveraging" -> "Macro"|> },
TrainingStoppingCriterion -> <|"Criterion" -> "Loss", "AbsoluteChange" ->
0.001|>]
Out[45]=
```

The final results of the training phase are depicted in Figure 10-25.



**Figure 10-25.** Net results of LeNet training

Extracting the trained model and appending the net encoder and decoder is done because the trained net does not come with an encoder and decoder at the input and output ports.

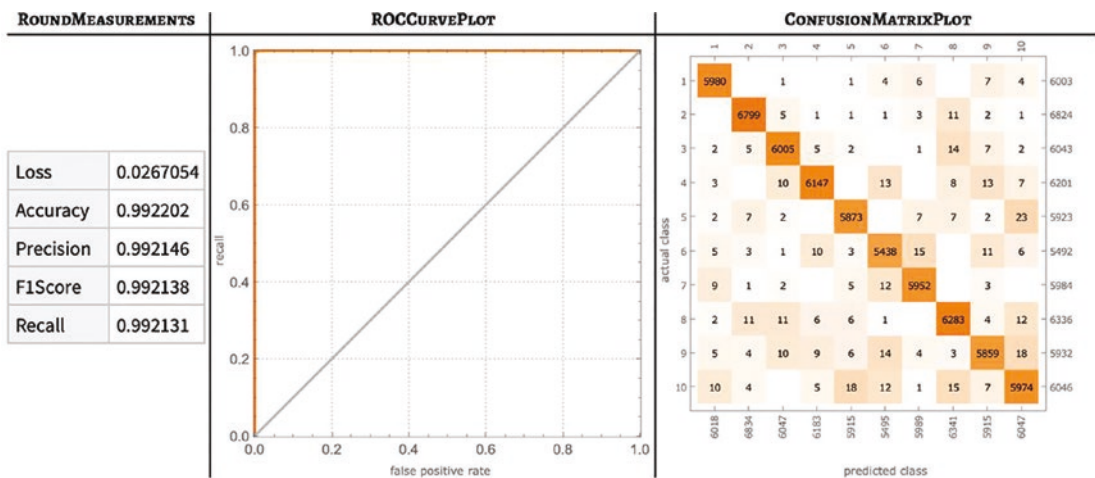
```
In[46]:=NetExtract[netResults["TrainedNet"],"LeNet NN"];
trainedLeNet=NetReplacePart[%,{"Input"->enc,"Output"->dec}];
```

## LeNet Model Assesment

The following grid (see Figure 10-26) shows the tracked measurements and plots of the training set. The measurements of the training set are in the RoundMeasurements property. To get the list of the values in each round, use RoundMeasurementsLists.

The performance of the training set is assessed with the round measurements, and the test set is evaluated with the validation measurements. Also, the ROC curves and the confusion matrix plot are shown in both cases.

```
In[48]:= netResults["RoundMeasurements"][[1 ;; 5]];
Normal[netResults["RoundMeasurements"][[6 ;; 7]]];
Grid[{{Style["RoundMeasurements", #1, #2], Style[%[[1, 1]], #1, #2],
      Style[%[[2, 1]], #1, #2]}, {Dataset[%], %[[1, 2]], %[[2, 2]]}},
Dividers -> Center] &[Bold, FontFamily -> "Alegreya SC"]
Out[50]=
```



**Figure 10-26.** Training set measurements

To see how the model performed on the validation set (see Figure 10-27), see ValidationMeasurements. To get the list of the values in each round, use ValidationMeasurementsLists.

```
In[51]:= netResults["ValidationMeasurements"][[1 ;; 5]];
Normal[netResults["ValidationMeasurements"][[6 ;; 7]]];
Grid[{{Style["ValidationMeasurements", #1, #2],
      Style[%[[1, 1]], #1, #2], Style[%[[2, 1]], #1, #2]}, {Dataset[%], %[[1,
      2]], %[[2, 2]]}}, Dividers -> Center] &[Bold, FontFamily -> "Alegreya SC"]
Out[53]=
```

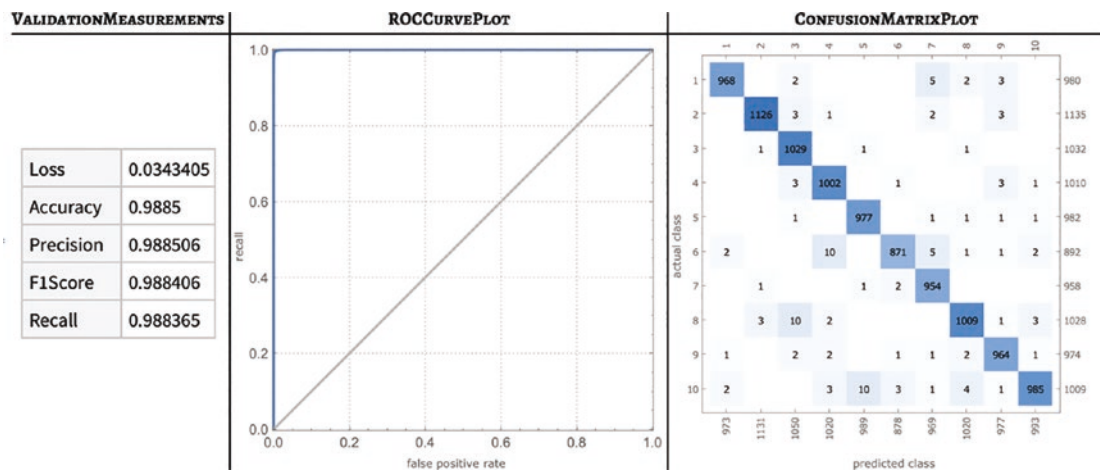


Figure 10-27. Validation set measurements

# Testing LeNet

Having finished the training and reviewed the round and validation measures, you are now ready to test the trained LeNet neural network with some difficult images to see how it performs (see Figure 10-28).

```
In[54]:=expls=Keys[{testData[[2150]],testData[[3910]],testData[[6115]],testData[[6011]],testData[[7834]]}]
Out[54]=
```

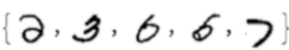


Figure 10-28. Difficult examples from the MNIST test set

The selected images belong to the numbers 2, 3, 6, 5, and 7.

```
In[55]:= trainedLeNet[expls,"TopProbabilities"]
Out[55]= {{2->0.999397},{3->0.999856},{6->0.906024},{6->0.990975},{7->0.999853}}
```

Write all of the results with the top probabilities with TableForm.

```
In[66]:= TableForm[Transpose@{trainedLeNet[expls,{"TopDecisions",
2}],TrainedLeNet[expls,{"TopProbabilities",2}]},TableHeadings->
{Map[ToString,{2,3,6,5,7},1],{"Top Decisions","Top Probabilities "}},
TableAlignments->Center]
```

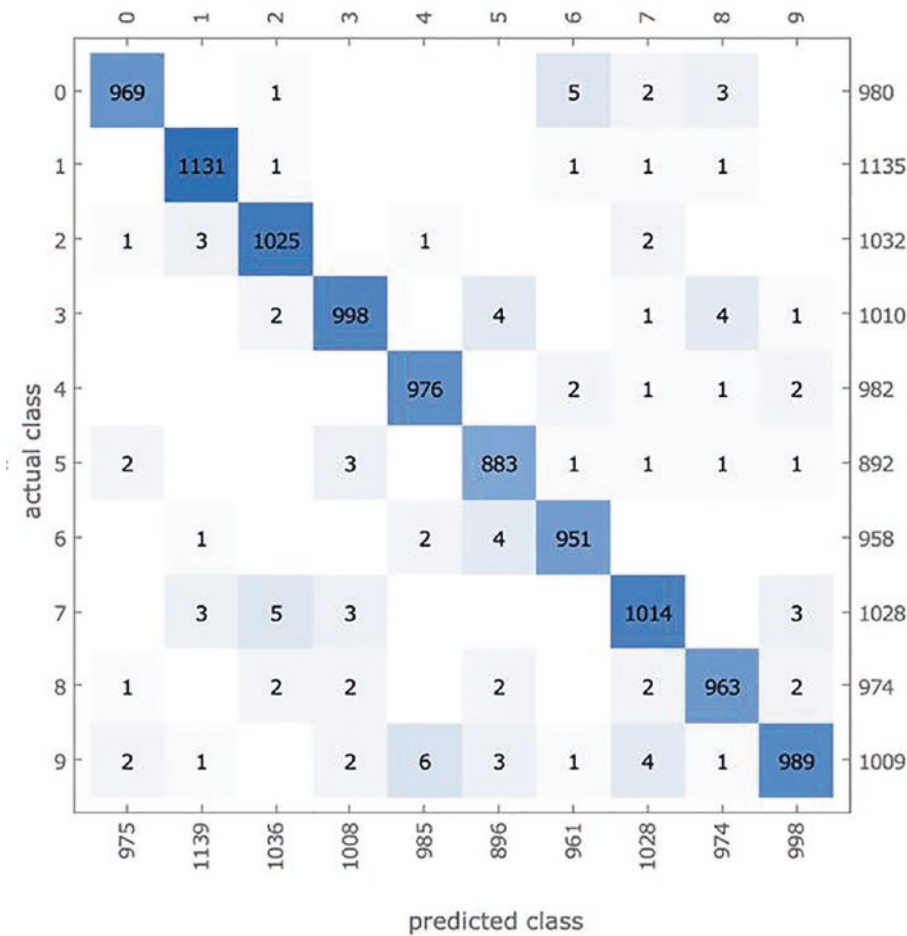
```
Out[66]//TableForm=
```

	Top Decisions	Top Probabilities
2	3	3->0.000580186
	2	2->0.999397
3	9	9->0.0000792077
	3	3->0.999856
6	0	0->0.0904324
	6	6->0.906024
5	5	5->0.00699159
	6	6->0.990975
7	3	6->0.990975
	7	7->0.999853

The trained net has misclassified the image of the number 5 because the top decisions are either a 5 or a 6, being 6 with top probability, which is wrong. Also, you can see the probabilities of the top decisions. Another form to evaluate the trained net in the test set is using NetMeasurements to set the net model, test set, and the interested measure. In the example, the measure of interest is the ConfusionMatrixPlot (see Figure 10-29).

```
In[67]:= NetMeasurements[trainedLeNet,testData,"ConfusionMatrixPlot"]
```

```
Out[67]=
```



**Figure 10-29.** *ConfusionMatrixPlot from NetMeasurements*

## GPT and LLM Basics

This section explores the neural network GPT models available in the Wolfram Language. You learn the basics of generative pre-trained transformers (GPT), the architecture of some GPT models inside Mathematica, and new LLM (large language model) Mathematica features.

## A Brief Overview

GPT is a series of AI models that uses deep learning and transformer architecture to generate human-like text by analyzing preceding text. LLM is a broader category encompassing models trained to understand and generate human-readable text. GPT models fall under the LLM category, representing just one kind of model within the broader LLM framework.

## LLM in the Wolfram Language

The Wolfram Language offers several new LLM-based functionalities, including the following.

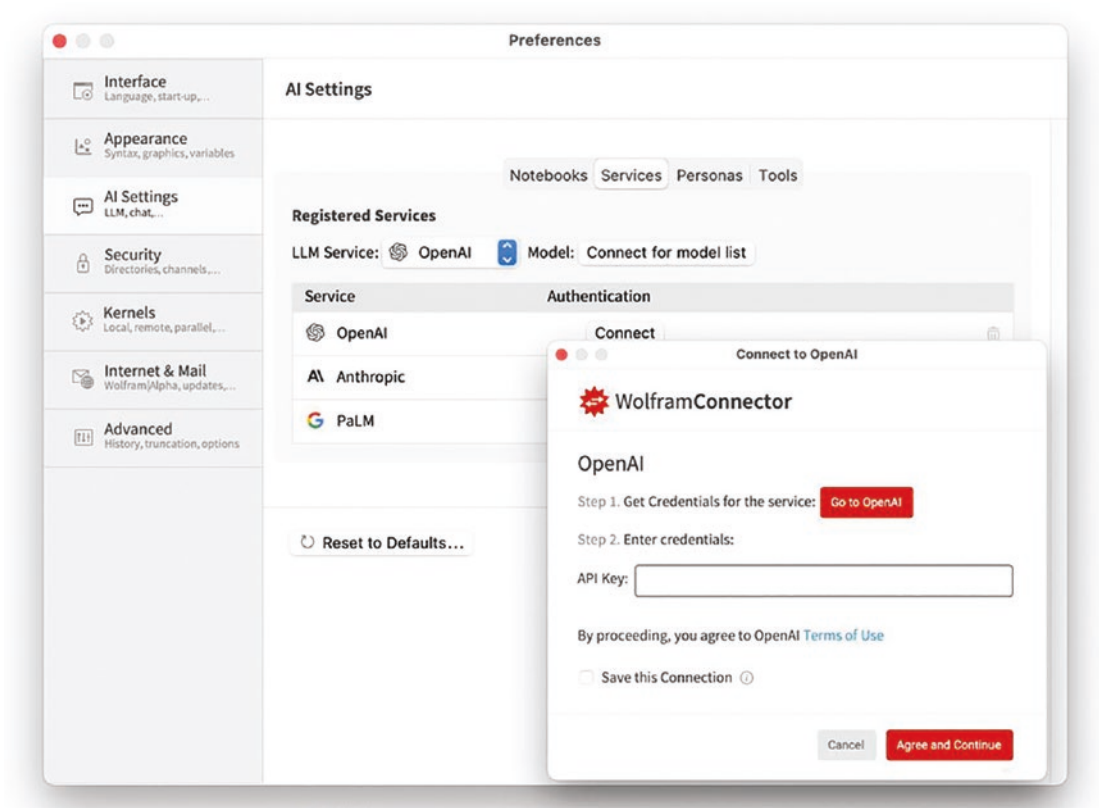
- **Chat Notebooks:** a new feature enabling efficient and accessible conversations with LLM (GPT-3, among others) like a traditional Mathematica notebook
- **Wolfram Prompt Repository:** a collection of useful prompts made by a community for easy access to LLM scope applications
- **LLM Function Integration:** seamless incorporation of LLM functions within Mathematica
- **GPT-1 and GPT-2:** available from the Wolfram Neural Net Repository

---

**Note** For LLM services in Mathematica, external API access is needed. Ensure your API key is valid; for example, for OpenAI, an active Chat GPT account with billing details is required. Be aware that API costs are separate from their subscription plans and vary based on the model used. Make sure to read OpenAI documentation for pricing and account details.

---

To connect to OpenAI GPT services, you first need to establish a connection. The most direct path to connect is through the settings or preferences section. Select the AI settings option from there, which shows various tabs related to chat notebooks, services, personas, and tools. The default tab has the general setting for the persona, LLM service, and temperature (model creativity), among other settings. To proceed, go to the Services tab and click Authentication, followed by Connect. This triggers a WolframConnector pop-up that requests the key access, as shown in Figure 10-30.



**Figure 10-30.** AI settings to connect LLM service from Mathematica

To get started, enter the key, save it by clicking the checkbox, and agree on the terms of use. Once linked, a checkmark appears under Authentication, like Figure 10-30. If a valid API is not linked, LLM services won’t work. To remove the key, click Disconnect and repeat the previous steps.

---

**Note** For quick API and LLM support, visit <https://support.wolfram.com/>

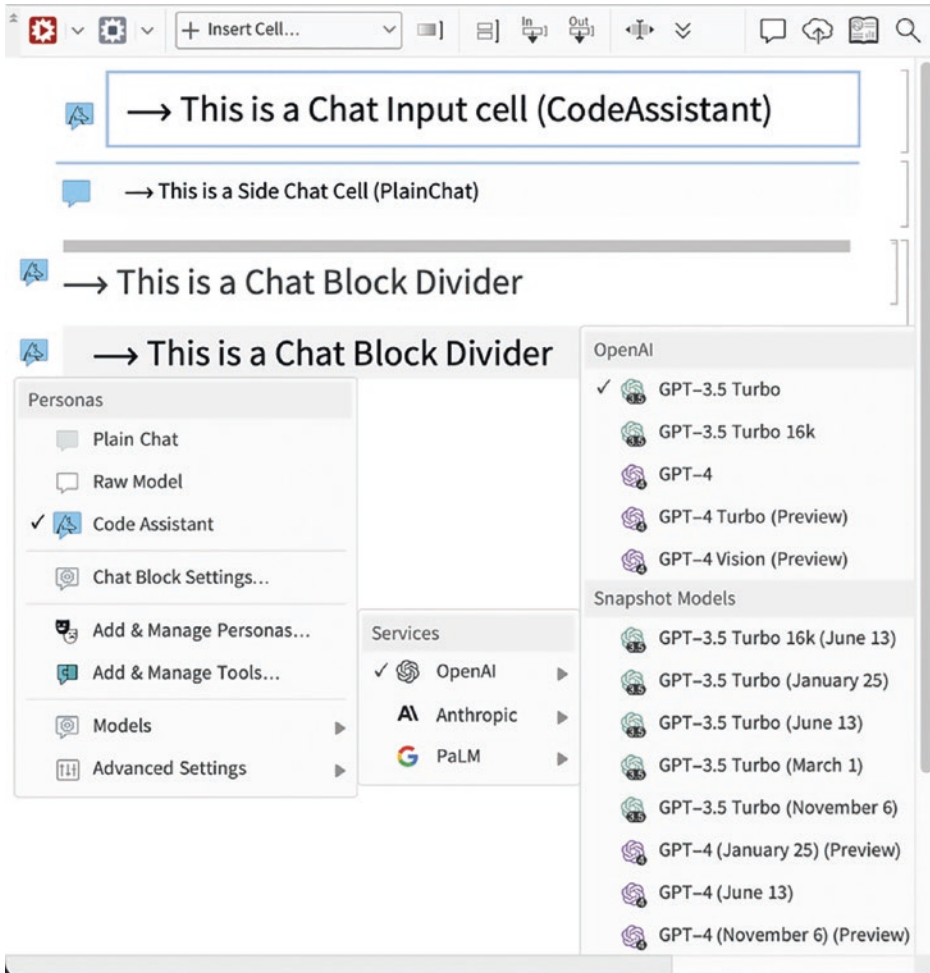
---

## Chat Notebooks

New types of notebooks have been developed apart from regular notebooks. These notebooks are specialized for LLM tasks. These are Chat-Enabled and Chat-Driven Notebooks. To create a new one, go to File ► New, then select Chat-Enabled or Chat-



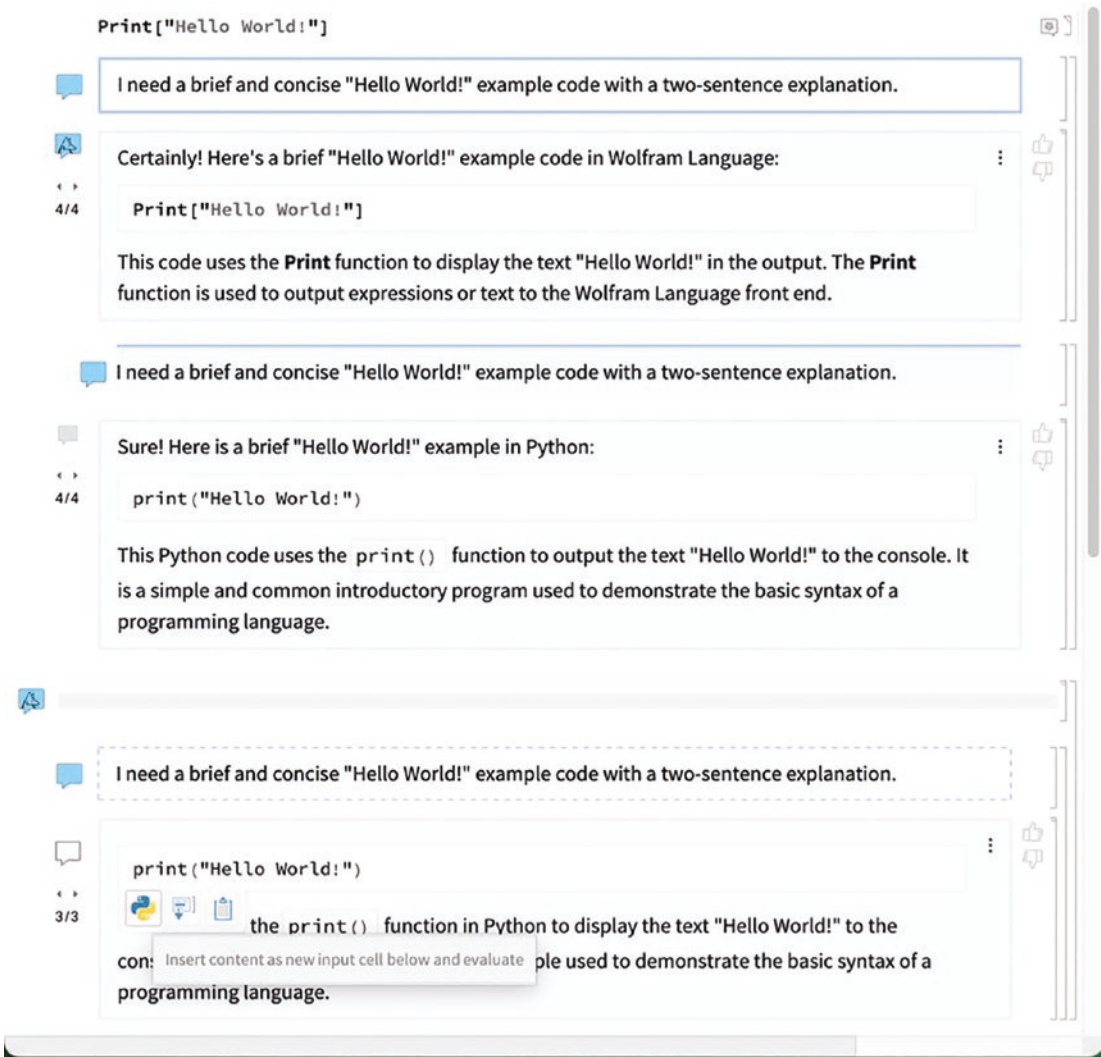
Driven Notebook. By default, chat-enabled use input chat cells with the code assistant persona (sets the LLM's response style), while chat-driven cells use PlainChat (basic dialog, no Wolfram code execution), as seen in Figure 10-31.



**Figure 10-31.** Multiple Chat cells and OpenAI available models

Apart from the different cells, Figure 10-31 show the various personas and GPT models for use. You can select the one that fits your needs. The base model version used in the following examples is with GPT-3.5 Turbo.

To create a new chat cell, press (') once. Press it twice for a side chat and three times for chat system input. To enable it in a regular notebook, click the chat cell icon in the top right corner (see Figure 10-31). Select “Enable AI chat features” to activate. Select the “Do automatic result analysis” option for LLM tips on output code. Try the example shown in Figure 10-32 to see if everything is working.



**Figure 10-32.** Sample prompt and output for CodeAssistant, PlainChat, and RawModel

In Figure 10-32, a chat icon is visible in the right cell bracket; this option lets you use LLM with Wolfram code like you use it in Mathematica. The chat history is sequential, and the conversation history output can also be accessed using the chat arrows. Side chat cells or blocks/delimiters separate chats. Distinct personas yield different responses; the CodeAssistant chat implies prompts in Wolfram code, whereas the Plain and RawChat yield output but do not imply that it's related to Wolfram code (unless specified in the prompt), resulting in Python code being used instead. Hovering over the code part allows you to either insert it as a newly evaluated cell, insert it, or copy it.

---

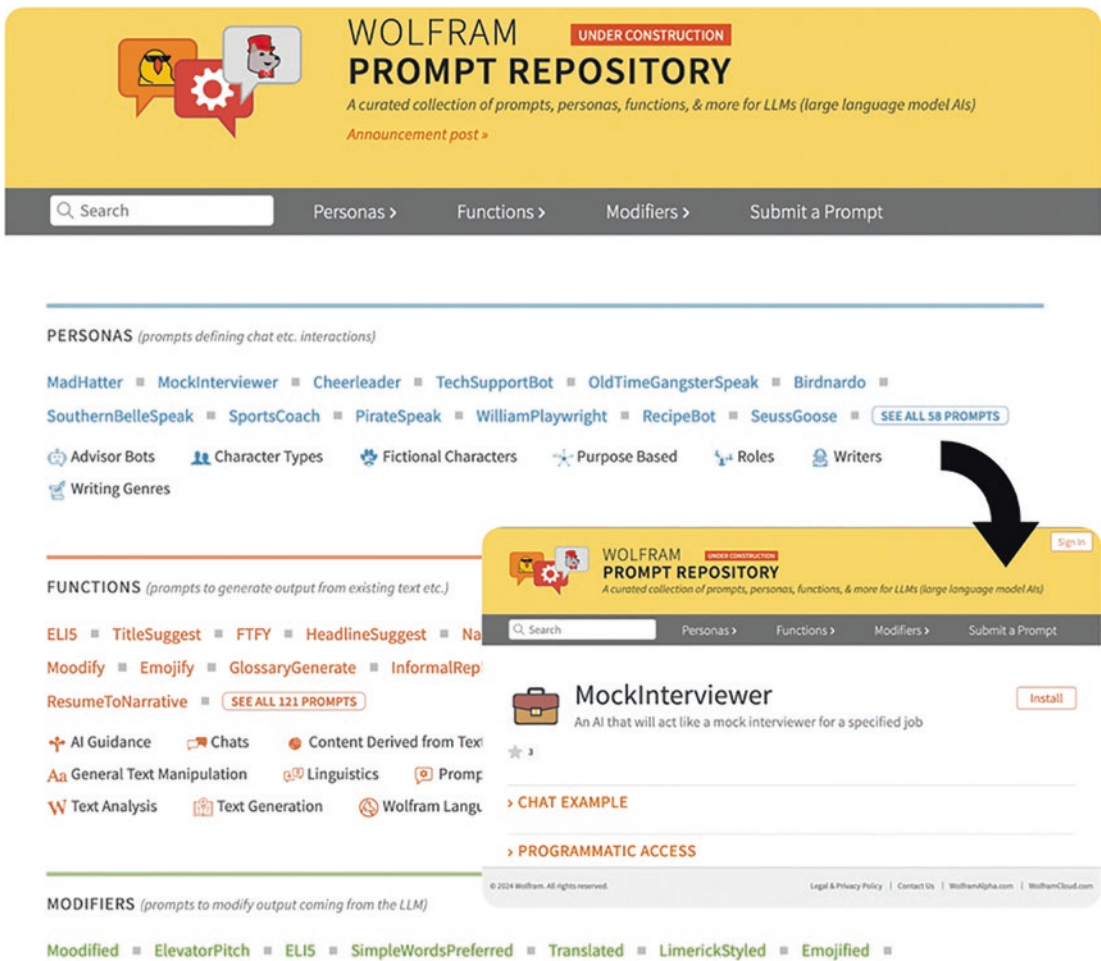
**Note** Keep your prompts concise; always verify the chosen model to avoid unexpected fees since models have different costs based on token count.

---

Chat cells can rerun the prompt and regenerate the response. But remember that the LLM prompts are not run by Mathematica kernel, so history is saved on the notebook. So, closing the notebook does not erase the conversation.

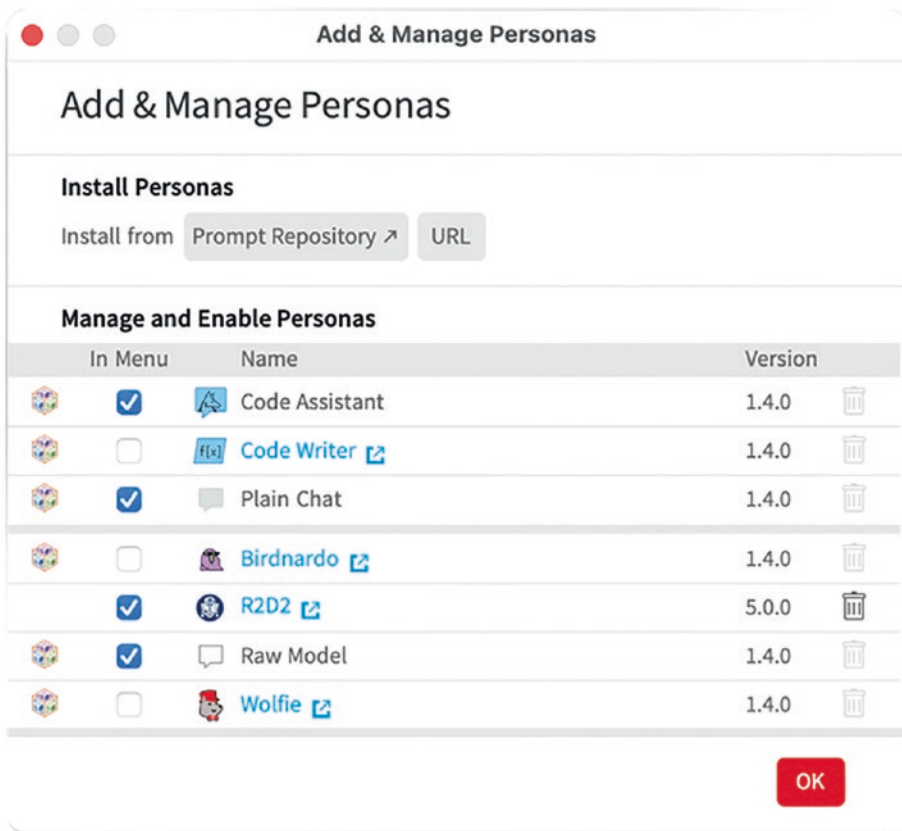
## Wolfram Prompt Repository

The Wolfram Prompt Repository gives you access to a large, curated base of prompts, from LLM prompts, personas, and costume functions. Navigating is similar to other repositories. Select from the accessible sections to find your desired prompts or persona for costume-style conversations. Once a prompt is selected various options are available, like chat samples and how to use it inside Mathematica. The platform further supports uploading, downloading, and utilizing various LLM components, as Figure 10-33 shows.



**Figure 10-33.** Wolfram Prompt Repository with the MockInterviewer prompt page

For instance, you can format output with different personas; select the persona from the drop-down menu (see Figure 10-31). To download a persona, go to the Personas tab in the AI setting and install via the prompt repository (see Figure 10-33) or enter the persona URL. Once installed, it should be available as depicted in Figure 10-34; this can also be done via Add & Manage Personas.



**Figure 10-34.** The Add & Manage Personas screen shows the R2D2 persona selected

Apart from personas, a combination of prompt modifiers can be used. These act on the input or output of a prompt. So, to invoke an input persona, use the character '@persona.' To call for a function input modifier, use '!prompt'; to call an output modifier, use '#param'; input and output modifiers go at the beginning and end of the prompt. To insert parameters to function modifiers, use the vertical bar to separate, like '#prompt|param' as defined in Figure 10-35.

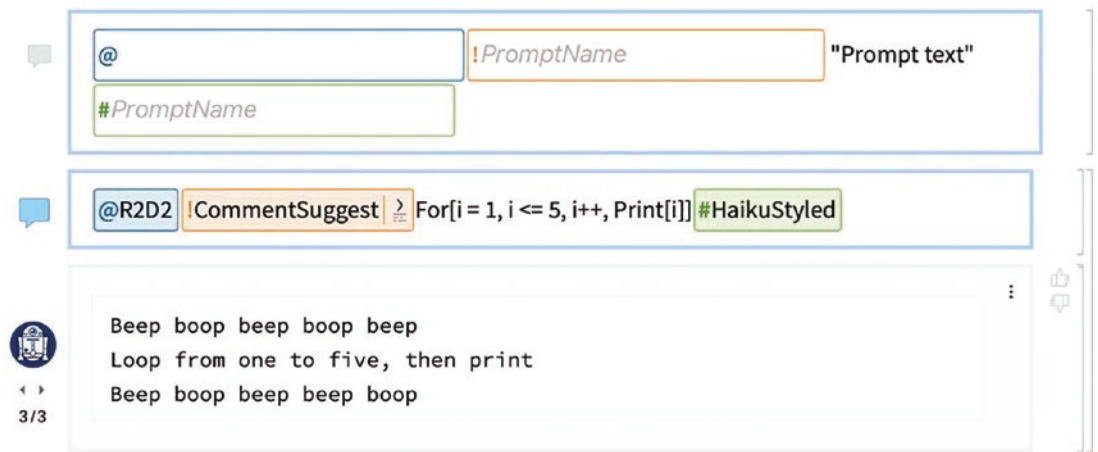


Figure 10-35. R2D2 code comment in Haiku style

## LLM Functionalities

Chat objects are used along with chat evaluate to manage LLM conversations within Mathematica. The chat object provides a convenient interface for interacting with the LLM and managing conversations in a notebook environment. What happens is that internally, LLM commands work as synthetic functions, which allows the LLM model to access the Wolfram tools (see Figure 10-36).

```
In[68]:= ChatEvaluate[ ChatObject[], "Break down this code in 3 simple
points? For[i=1,i<=5,i++,Print[i]", LLMEvaluator -> <|"Prompts" ->
{LLMPrompt["ELI5"]}]|>] (*Explain Like I'm Five*)
Out[68]=
```

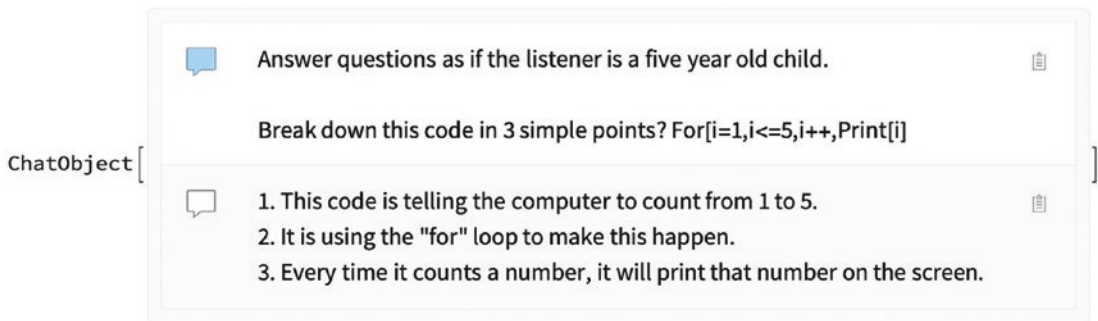


Figure 10-36. Chatobject for an LLM text prompt

To retrieve the chat contents and tokens, use the words “Messages” and “Usage”.

```
In[69]:=
%["Messages"]
%%["Usage"]
Out[2]= {<|"Role" -> "User", "Content" -> "Answer questions as if the
listener is a five year old child. Break down this code in 3 simple points?
For[i=1,i<=5,i++,Print[i]", "Timestamp" -> DateObject[{2024, 2, 22,
11, 22, 5.627397}, "Instant", "Gregorian", -6.], "Annotations" -> <|{1,
129} -> "Prompt">|>, <|"Role" -> "Assistant", "Content" ->
"1. This code is telling the computer to count from 1 to 5.
2. It is using the \"for\" loop to make this happen.
3. Every time it counts a number, it will print that number on the \
screen.", "Timestamp" -> DateObject[{2024, 2, 22, 11, 22, 6}, "Instant",
"Gregorian", -6.], "Annotations" -> <|{1, 184} -> "Completion">|>}}
Out[70]= 92 tokens
```

Like the previous example, you can set a prompt with a specific configuration with `LLMConfiguration` evaluated with `LLMEvaluator`, like the base model, temperature, stop tokens, and so forth. It can also be used to generate text (`LLMSynthesize`), retrieve text (`LLMPrompt`), or use a template function (`LLMFunction`), as the following code shows.

```
In[70]:= llmConfig = LLMConfiguration[<|"Prompts" -> LLMPrompt["ELI5"],
"Model" -> "GPT-3.5-Turbo", "Temperature" -> 0.1, "MaxTokens"
-> 5|>]; LLMSynthesize["Break down this code in 3 simple points?
For[i=1,i<=5,i++,Print[i]", LLMEvaluator -> llmConfig]
Out[71]= Sure! Here's a
```

---

**Note** The default LLM configuration is in `$LLMEvaluator` but can be overridden.

---

```
In[72]:= $LLMEvaluator=LLMConfiguration[<|"Prompts"-> LLMPrompt["ELI5"],
"Model"->"GPT-3.5-Turbo", "Temperature"->0.1,"MaxTokens"-> 5|>]
Out[72]= LLMConfiguration[Model: <|Service->Automatic,Name-
>GPT-3.5-Turbo|>]
```



## GTP-1 and GPT-2 Models

Besides external LLM services, open models like GPT-1 and GPT-2 are accessible in Mathematica. These models are predecessors to recent GPT models. GPT-1 is one of the initial models trained on a large book dataset, and GPT-2 is an improved version of GPT-1, trained on the WebText dataset. Let's look at some information about GPT-1 and GPT-2; note that the output here is truncated, given the large text.

```
In[72]:= Row[{Short[NetModel[ "GPT Transformer Trained on BookCorpus
Data", #] & /@ {"Details", "ShortName"} // Column, 4], Short[NetModel[
"GPT2 Transformer Trained on WebText Data", #] & /@ {"Details","ShortName"}
// Column, 4]]]
```

Out[72]= Released in 2018, this Generative Pre-Training Transformer (GPT) model is pre-trained in an unsupervised fashion on a large corpus of English text. This model can be further fine-tuned with additional output layers to create highly accurate NLP models for a wide range of tasks. It uses bi-directional causal self-attention, often referred to as a transformer decoder.

GPT-Transformer-Trained-on-BookCorpus-Data

Released in 2019, this model improves and scales up its predecessor model. It has a richer vocabulary and uses BPE tokenization on UTF-8 byte sequences and additional normalization at the end of all of the transformer blocks.

GPT2-Transformer-Trained-on-WebText-Data

You can try to retrieve other data, like in the LeNet example. Let's look at model variants and task types examples.

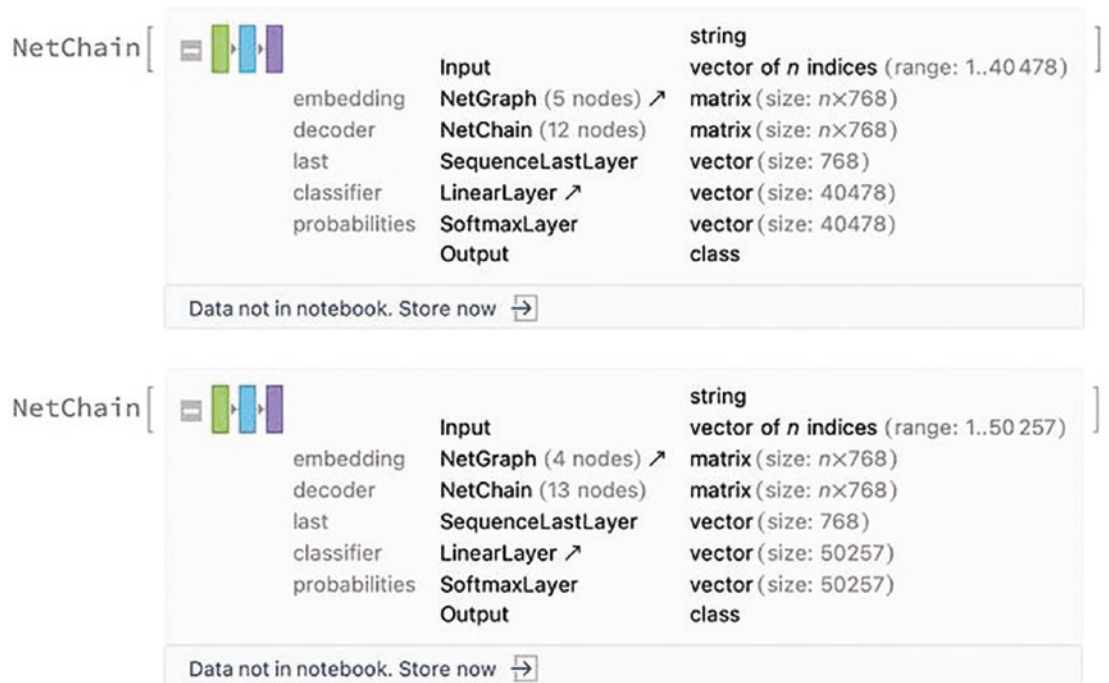
```
In[73]:= NetModel["GPT Transformer Trained on BookCorpus Data", #] & /@
{"ParametersAllowedValues", "Variants"}
NetModel["GPT2 Transformer Trained on WebText Data", #] & /@
{"ParametersAllowedValues", "Variants"}
Out[73]= {<|Task->{FeatureExtraction,LanguageModeling}|>,{GPT Transformer
Trained on BookCorpus Data,Task->FeatureExtraction},{GPT Transformer
Trained on BookCorpus Data,Task->LanguageModeling}}
Out[74]= {<|Task->{FeatureExtraction,LanguageModeling},Size-
>{117M,345M,774M}|>,
```



```
{GPT2 Transformer Trained on WebText Data,Task->FeatureExtraction,Size->117M},
{GPT2 Transformer Trained on WebText Data,Task->FeatureExtraction,Size->345M},
{GPT2 Transformer Trained on WebText Data,Task->FeatureExtraction,Size->774M},
{GPT2 Transformer Trained on WebText Data,Task->LanguageModeling,Size->117M},
{GPT2 Transformer Trained on WebText Data,Task->LanguageModeling,Size->345M},
{GPT2 Transformer Trained on WebText Data,Task->LanguageModeling,Size->774M}}}
```

As seen in the output, variants have different task types and a specific number of parameter sizes, like 117M, 354M, and 774M million parameters. You can pick a model by specifying the parameters, for instance, picking the language-trained model and trying to generate text based on the prediction of the next token (see Figure 10-37).

```
In[75]:= gpt1=NetModel[{"GPT Transformer Trained on BookCorpus
Data","Task"-> "LanguageModeling"}]
gpt2=NetModel[{"GPT2 Transformer Trained on WebText Data","Task"->
"LanguageModeling"}]
Out[75]=
```



**Figure 10-37.** GPT-1 and GPT-2 embedded architectures

For the token function, the input parameters are the initial text, token count (default 10), and temperature (default 1). In simple terms, this function samples predictions. It attaches each new token to the original string for the fixed token count and returns the initial text plus the generated tokens text.

```
In[76]:= generateText[LLmodel_][initialText_, tokenCount_ :
10, temperature_ : 1] := Fold[StringJoin[#1, LLmodel[#1, {"RandomSample",
"Temperature" -> temperature}]] &, initialText, Range[tokenCount]]
```

Where a token refers to a unit of text that the model reads. It can be as short as one character or as long as one word, like “a” or “app.” The model looks at these tokens individually to understand and generate text based on them. So, for GPT-2, BPE tokenization is a method used to break down words into smaller parts.

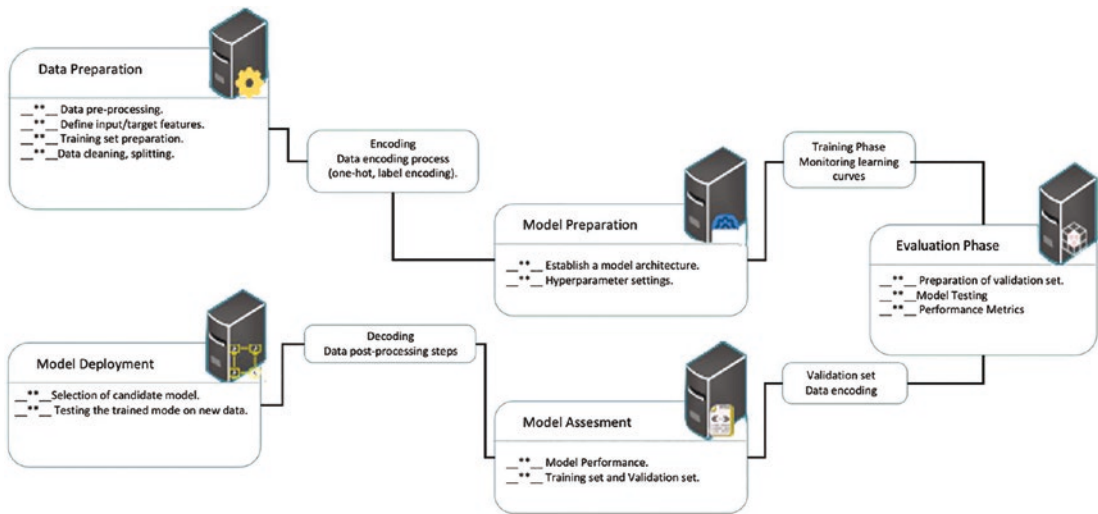
```
In[77]:= generateText[gpt1]["Alan Turing was a British mathematician
and logician who is considered a pioneer in the field of computer
science.",20,0.5]
Out[77]= Alan Turing was a British mathematician and logician who is
considered a pioneer in the field of computer science.he is a physicist and
is a very good scientist .
he is also a friend of george w.
```

```
In[78]:= generateText[gpt2]["Alan Turing was a British mathematician
and logician who is considered a pioneer in the field of computer
science.",20,1]
Out[78]= Alan Turing was a British mathematician and logician who is
considered a pioneer in the field of computer science. At the time of his
independence he was selected to pen one of the first (173) contributions to
```

As seen comparing both responses, there is still room for improvement. GPT-1 output seems incoherent with unrelated elements. In contrast, GPT-2 shows more context talking about Turing’s career but still lacks clear, complete sentences.

## Final Remarks

In summary, the following road map for the general schematics, construction, testing, and implementation of a machine learning or a neural network model within the Wolfram Language scheme are in Figure [10-38](#).



**Figure 10-38.** *Model overview for training and testing*

The diagram shows a route that can be followed directly; despite this, there may be intermediate points between each process within the route since the route may vary depending on the type of task or problem being solved. However, the route focuses on exposing the important and general points to construct a model using the Wolfram Language. Within the data preparation phase are previous processes, such as data integration, the type of data collected (structured or unstructured), transformations in the data, cleaning in data modules, and so on. So before moving on to the next phase, there must be a pre-processing of the data, to have data ready to be fed to the model.

Model preparation covers aspects such as the choice of the algorithm or the methods to use, depending on the type of learning; establishing or detecting the structure of the model; and defining the characteristics, input parameters, and type of data that is used, whether it be text, sound, numerical data, and tools to be used. All this is linked to a process called feature engineering, whose primary goal is to extract valuable attributes from data. This is needed to move on to the next point, the training phase.

The evaluation phase and model assessment consists of defining the evaluation metrics, which vary according to the task or problem being solved, and preparing the validation used later. The model's output is converted back to a clear, interpretable format at the decoding phase, readying for practical use. At this point, it is necessary to

emphasize that the preparation of the model, training, evaluation, and assessment can be an iterative process, including tuning of hyperparameters, adjustments on algorithm techniques, and model configurations such as internal model features. The purpose is to establish the best possible model capable of delivering adequate results and finally reaching the model deployment phase, which defines the model chosen and tested on new data.

# Index

## A

AccuracyRejectionPlot, 336  
Activation functions, 368, 370, 372, 427  
Algebraic equations, 34–37  
Algebraic expressions, 33, 34  
Algorithm specifications tooltip, 330  
Alphas, 307  
AND operator, 32, 36  
Arithmetic mean, 239, 341  
Arithmetic operations, 18, 89–91  
ArrayPath, 399  
Arrays  
    arrangements, 69  
    Array command, 68  
    characteristics, 70  
    ConstantArray function, 69  
    data array, 69  
    F function, 69  
    MatrixForm, 71  
    sparse array, 71  
    SparseArray command, 70  
ArraysCount, 395  
AspectRatio, 190  
Assigning values to variables, 19–21  
Assistance  
    autocomplete pop-up menu, 50, 51  
    documentation, 51  
    functions output, 52, 53  
    Head command, 52  
    RandomPolygon function, 51  
Associations

    Association command, 101  
    associations, 102  
    AssociationThread, 102  
    complex structures, 103  
    entries, 101  
    error, 102  
    forms, 101  
    position, 101  
    semicolon, 101  
    uses, 101  
    values and keys, 102  
AtomQ function, 26, 33

## B

Bar chart grid, 244  
Bar graphs, 242, 243  
Basic plotting  
    AxesLabel option, 29, 30  
    cubic plot, 27, 28  
    dashed tangent function, 29  
    multiple functions, 28  
    PlotLabel option, 29  
BatchEvaluationSpeed, 355  
BlockRandom, 235  
Boolean operators, 31, 32  
Boston Homes data, 309  
Boston Homes price dataset, 309  
Box plot, 252, 253  
    BoxWhiskerChart, 252  
    multiple, 254  
    Whiskers, 253

## INDEX

Box whiskers plot, 294, 298, 299  
Built-in functions, 22, 23, 32  
Business and Minimal plot themes, 229

## C

CapsNet, 401  
CapsNet neural net model, 402  
Ceiling function, 63  
ChartElementFunction option, 258  
Chart Element Schemes  
    palette, 256, 257  
Chart Type, 258, 259  
Chat-Driven Notebooks, 438  
ClassAveraging, 431  
Classifier function, 332  
ClassifierFunction object, 328  
ClassifierMeasurements, 332  
ClassifierMeasurementsObject, 333  
Classify command, 327  
Cluster classification model, 354, 356, 357  
ClusterClassify, 353  
ClusteringCompnents, 343  
Clusters, 340  
CodeAssistant chat, 441  
Code efficiency, 41  
Code performance, 42, 43  
ColorData object, 217  
ColorFunction, 222, 227  
Color palette, 216  
Column graphics, 212  
Combining plots  
    Column command, 208  
    cosine and sine plot, 207  
    graphic objects, 206  
    graphics, 206  
    graphs, 206  
Comma-separated value (CSV),  
    148–150, 275

Complex numbers, 57  
Computations  
    FullForm, 48  
    HoldForm command, 49  
    InputForm, 47  
    StandardForm, 47  
    Trace command, 49, 50  
    TreeForm, 47–50  
ComputeUncertainty, 317  
ConfusionMatrixPlot, 334, 435, 436  
ContentElements property, 280  
Contour lines, 223  
ContourPlot command, 222  
CrossEntropyLossLayer, 373, 430  
Cross-hatching fillers, 204  
Customized 3D plot, 220  
Customizing plots  
    labeled axes and functions, 198  
    PlotLabel, 197  
    3D plots, 197  
    Wolfram Language, 197

## D

Databases, 184, 186  
Data clustering method, 338  
Data exploration  
    column's class and sex, 325  
    elements, 325  
Datasets, 103, 274, 303  
    adding values  
        associations, 116  
        AssociationThread, 114  
        column, 114, 115  
        ID column dataset, 114, 115  
        ReplacePart function, 116  
        row, 113  
    associations, 107  
    column headers, 105

- column name, 105, 106
- columns and values, 111, 112
- create, 109
- customization, 134
  - animal dataset, 135, 137, 138
  - Background option, 135
  - bold style, 136
  - ExampleData, 134
  - HiddenItems, 136
  - mixing colors, 135, 136
  - options, 136
  - suppressed rows and columns, 136, 137
- Dataset function, 104
- decimal form, 124
- dropping values, 117, 118
- filtering values
  - clear symbols, 122
  - count data, 121
  - filtered data, 120
  - grouping, 121, 122
  - pure functions, 120, 121
  - selected subjects, 120
  - tagged dataset, 119
- forms, 104
- joining/merging, 132–134
- labeled rows, 107
- list of association, 104, 106
- name/age, 110
- Normal function, 111
- numeric dataset, 122, 123
- position, 104
- quantities
  - DateObject, 162
  - ID/sales per day, 161, 162
  - magnitudes, 161
  - Normal command, 161
  - price column, 161
  - quantity type, 160
  - Rule command, 163
  - timeline, 163
- queries, 111
- reversed elements, 124, 125
- rows and columns, 108
- row selection, 107, 108
- select data, 109
- sorted data, 123, 124
- square root function, 125, 126
- Take function, 112
- uses, 103
- Values command, 109–111
- value selection, 108
- Data visualization
  - AspectRatio, 190
  - DateListPlot, 195
  - Date plot, 195
  - ListLinePlot, 195
  - logarithm scale, 187
  - Mathematica, 187
  - PlotRange, 188, 189
  - tools, 187
  - 2D plots, 187
  - types of graphs, 187
- Date and time
  - date format, 24
  - DateObject command, 23, 24
  - DatesString, 25
  - Natural language, 24
  - sunrise and sunset times, 24
  - TimeObject, 25
  - time zone, 24
- DateListPlot, 195
- 3D bar charts grid, 245
- Debugging, 45, 47
- Decoded house, 383
- Decoder, 380, 381

## INDEX

DensityHistogram, 259, 260  
Density plots, 225, 226  
Descriptive statistics  
    function, 287  
    grid view, 291  
    Iris data and computations, 287  
    TabView, 288  
    versicolor species, 292  
    Wolfram Language, 293  
DescriptiveStats variable, 289  
Digits, 60, 61  
Dimensionality problems, 345  
DimensionReducerFunction, 345  
DimensionReduction, 345  
Dispersion measurements, 240  
DistanceFunction, 340, 349  
Distribution chart plot, 293  
Documentation, 50, 51

## E

EchoTiming function, 43  
ElementwiseLayer, 368, 369, 371, 374  
ElemewiseLayer, 375  
Encoder, 375, 377  
    Boolean type NetEncoder, 375  
    Class type, 376  
    and decoder, 382  
    NetEncoder, 376  
Encoding, 375, 382  
Equivalent operator, 32  
EuclideanDistance, 340, 355  
ExampleData, 346  
Exponentials, 62  
Export  
    CarStoppingDistance, 171, 172  
    ColumnDescriptions, 171  
    ContentObject function, 177

    CSV format, 170, 172  
    DAT format, 169  
    ExampleData command, 170  
    Export command, 167  
    file extensions, 179  
    FileNames command, 178  
    files, 170, 173  
    FindFile, 178  
    JSON format, 174–176  
    list of names, 170  
    NotebookDirectory command,  
        167, 178  
    prime numbers, 167  
    SetDirectory, 167, 178  
    sheets name, 168  
    SystemOpen, 168  
    tables, 169  
    tabular data, 168  
    TVS format, 170  
    working directory, 167  
    XLS format, 173  
    XLSX format, 173  
\$ExportFormat, 166  
Expressions, 19  
External connections, 179, 180  
External resources  
    arrow function, 182  
    custom functions, 182  
    FindExternalEvaluators[“NodeJS”], 182  
    Math.sqrt, 182  
    node.jsfunction, 183  
    registered key, 182  
Extracted trained net, 406

## F

Factorial, 63, 123  
File operations, 184, 186



Filled horizontal style, 204  
 Filled plots, 203  
 FindClusters function, 338, 342  
 Fisher's dataset, 284, 285  
 Fisher's Irises data, 277  
 Fisher's Irises dataset object,  
     274, 279, 344  
 Flattened chain, 387  
 Floor function, 63  
 Framed ListPlot, 201  
 Frame option, 200  
 FromDigits function, 61  
 FunctionLayer, 374, 375  
 Functions, 122  
     column/row  
         ceiling function, 127  
         DeleteDuplicates function, 129–131  
         DuplicateFreeQ function, 130  
         GroupBy, 132  
         MapAt function, 128  
         output, 127, 128

## G

Generative pre-trained  
     transformers (GPT)  
         AI models, 437  
         architecture, 436  
         LLM category, 437  
 GPT-1 and GPT-2 embedded  
     architectures, 447  
 Gradient color, 217  
 Gradient descent algorithm, 303, 307  
 Gradient technique, 205  
 GraphicsColumn, 213  
 GraphicsGrid, 213, 214, 251  
 GraphicsRow, 213  
 Grid command, 67

Gridded plot, 201  
 GridLines command, 202, 222

## H

Handling errors, 43, 44  
 Hash tables, 138  
     add row, 145  
     assigned value, dataset, 144  
     associations, 139, 140  
     dataset representation, 141, 142  
     framed levels, keys, 142, 143  
     graphic representation, 139  
     KeyDrop, 144  
     KeyExistQ, 143  
     KeyMemberQ, 143  
     Keys command, 140  
     KeySelect, 144  
     KeyTake, 144  
     Lookup, 143  
     MapIndexed, 140, 141  
     operations, 140  
     User key, 141  
     uses, 139  
 Histograms, 245, 261  
     density, 258  
     origin, 246  
     origins, 248  
     PairedHistograms, 247  
     PDF and CDF plots, 249  
     random real numbers, 246  
     shapes grid, 247  
     variable classes, 245  
     for versicolor, 295  
 Hue color function, 220  
     colored Hue values, 221  
     3D scatter plots, 221  
 Hyperbolic cosine plot, 191

## I

imensionReductionFunction, 345  
 Import command, 148  
 Infix notation, 30, 59  
 Input Assistant, 50  
 IntegerDigits function, 60  
 Integers, 56  
 Integrated Global Radiosonde  
     Archive (IGRA), 156  
 InterquartileRange function, 242  
 Iris data, 282

## J

JavaScript Object Notation (JSON)  
     files, 153–156

## K

k-means method, 343  
     classifier information, 354  
     elements, 344  
     FindClusters, 347  
     principal components, 348  
     vmathematical foundation, 344  
 k-means technique, 353

## L

Large language model (LLM)  
     AI settings, 438  
     API, 438  
     Chatobject, 444  
     framework, 437  
     GPT-1 and GPT-2, 446  
     LLMEvaluator, 445  
     parameter sizes, 447  
     scope applications, 437

    tasks, 438  
     Wolfram code, 441  
     Wolfram Language, 437  
     Wolfram tools, 444  
 Lasso regression name, 319  
 Learning curve, 307  
 LearningRateMultipliers, 362  
 LeNet, 423, 446  
     architecture, 425–427  
     MNIST dataset, 424  
     NetModel command, 423  
     neural network, 423, 434  
     RoundMeasurements, 432  
     training, 432  
     vNetGraph, 428  
     Wolfram Language, 428  
 LinearLayer object, 360, 361, 363, 364, 377  
 Linear model, 306  
 LinearModelFit command, 265  
 Linear regression model, 308, 312, 333  
     correlation matrix, 311  
     MEDV and RM scatter plots, 310  
     model creation, 310  
     predict function, 308  
 Linear relationship, 266  
 ListContourPlot, 224  
 ListLinePlot, 192, 193, 307  
 ListPlot, 191, 342  
 Lists, 55  
     alternatives, 87  
     Apply function, 92  
     ArrayReshape function, 83  
     arrays, 67  
     assigning/removing values  
         Append/Prepend, 80  
         ArrayPad function, 81, 82  
         Delete command, 80  
         Drop, 80

- Insert function, 81
  - item position, 79
  - new values, 80
  - one-sided terms, 81
  - Replace function, 80
  - Cases function, 84–87
  - commands, 92
  - computations, 91
  - conditional matching, 87
  - definition, 55, 56
  - elements, 65
  - Flatten function, 83
  - functions, 68
  - Grid command, 67
  - identifier/symbol, 65
  - increment/decrement operators, 68
  - iterator, 67
  - joining, 91
  - level of specification, 87
  - list of lists, 66
  - manipulation, 77
  - Map function, 92
  - nested lists, 71, 72, 82, 83
  - objects, 65
  - partitions, 82, 83
  - pattern shape, 84
  - Pick command, 84
  - PrimeQ function, 93
  - RandomChoice function, 85
  - random integers, 68
  - Range functions, 65
  - retrieving data
    - backward indices, 78
    - elements, 77
    - in-depth list, 79
    - index notation, 77
    - nested list, 78
    - Part function, 77
    - Rest function, 79
    - span notation, 78
    - Take function, 79
  - Reverse command, 82
  - Select command, 84
  - SortBy, 83
  - sorting, 82
  - TableForm, 67
  - Table functions, 65–67
  - underscore function, 86
  - to variables, 65
  - Logarithmic functions, 62
  - Logical operators, 30
  - Logistic regression, 321
    - DeleteMissing command, 323
    - learning curve and accuracy
      - curve, 330
    - missing data, 323
    - standard deviation and r-squared, 322
    - TopProbabilities, 331
    - variable, 321
  - LogProbabilities, 331
- ## M
- Machine learning, 314, 327, 357
    - gradient descent, 303
    - hyperparameter, 304
    - RandomReal function, 304
    - Wolfram Language, 303
  - Machine precision, 59, 60
  - Manhattan distance, 347
  - Mathematica, 218, 264, 265, 275, 301, 315, 369, 411, 441
    - capacity, 4
    - cells, 7
    - computation, 7
    - definition, 1

## INDEX

### Mathematica (*cont.*)

- expression, 19
- input, 7
- interface, 6
- kernel, 6
- notebook, 5, 7, 8
- precision, 18
- preferences window, 8
- structure
  - cells, 5
  - input types, 5, 6
  - notebook, 4
  - welcome screen, 3
- suggestion bar, 7
- uses, 2

### Mathematica version, 198

### Matrix

- definition, 73
- drawn lines, 74
- list of lists, 74
- MatrixQ, 74
- operations, 75
- restructuring, 76
- transpose, 74

### MatrixPlot, 311, 312

### Max and Min functions, 64

### Mean function, 240

### MeanSquare, 413

### Mean squared layer

- MeanSquaredLossLayer, 366
- NetEncoder, 367
- parameters, 365

### MeanSquaredLossLayer, 366, 413

### MeansSquaredLoss, 373

### Median function, 240

### MersenneTwister method, 235

### Mesh option, 192

### MethodOption, 318

### MNIST database, 424

### Model deployment phase, 450

### Model preparation, 449

### MultiAxisArrangement, 210

### Multiple plots, 197, 207

### MXNet format, 400

### MXNet framework, 361, 426

### MXNet network, 399

### MXNet operation, 359

### MXNet ops plot, 401

### MXNet symbols, 398

## N

### National Oceanic and Atmospheric Administration (NOAA), 156

### Negation operator, 32

### Nest command, 289

### Nested chain, 386

### Net2 classification plot, 410

### NetChain, 384–387, 392, 393

### NetChain NetCH2, 386

### Net classification plot, 407

### NetDecoder, 380

### NetEncoder, 376, 377

### NetEvaluationMode, 385

### NetExtract, 364

### NetFlatten, 387

### NetGraph command, 383, 388–394

### NetInitialize, 429

### NetMeasurements, 435

### NetPort, 390

### NetPortGradient, 365

### NetResultsObject, 415

### NetTrainResultsObjec, 406

### Neural network

- batch size, 408

- containers, 383
- data, 360
- layers, 359
- LeNet, 423
- linear layer, 360
- MaxTrainingRounds, 408
- model implementation, 406
- MXNet operation, 359
- Net2, 409
- NetChain, 384
- NetTrain, 403
- perceptron model, 404
- progress information panel, 405
- ResourceSearch, 420
- Standardize function, 404
- training data, 404
- WLNet format, 417
- Wolfram Language, 360, 403
- N function, 59
- NormalEquation, 319
- Notches, 254
- Notebook, 1, 4, 12
  - feature, 15–17
  - security, 53
  - style, 15–17
- UI, 9
  - abort options, 9, 10
  - application toolbar menu, 11
  - cell management functions, 10
  - code input cell, 11
  - extensive options, 9, 10
  - options, 10
  - prominent toolbar Ruler, 12
  - text cell options, 10
  - toolbar, 9
- NotebookObject, 277
- Number notations, 59
- Numeric expressions, 380

## O

- OptimizationMethod, 319
- Ordinary least squares method, 262
  - equations, 262
  - points, 263
  - summation, 262
- OR operator, 32, 36
- OrthantWiseNewton, 319

## P

- PaddingSize, 379
- Palettes, 14, 15
- ParameterConfidenceIntervalTable
  - property, 268
- ParameterTable property, 267
- Pattern matching, 84
- Pearson coefficient, 264
- Pie charts, 250, 326
- PlotMarkers, 193
- PlotStyle, 193
- PlotTheme, 227, 228
- Plotting commands, 196
- Plotting graphs, 191
- PoolingLayer, 379
- Predict function, 308, 315, 328
- PredictorFunction, 313
- PredictorMeasurements, 316
- PredictorMeasurementsObject, 317
- Prefix notation, 59
- Probability density functions (PDFs), 248
- ProbabilityHistogram, 336
- Pure functions, 95

## Q

- Quantity function, 297
- Quartile calculation, 241
- Query command, 286

## R

Ramp function, 369  
 Random data, 338  
 RandomInteger function, 233, 234  
 Random numbers, 123  
     BlockRandom function, 235  
     functions, 233  
     MersenneTwister method, 235  
     sublist, 234  
 RandomReal function, 23  
 Random sampling  
     expression, 236  
     RandomChoice function, 235  
     replacement, 237  
     weights and elements, 236  
 RandomSeeding, 339, 363  
 Rational numbers, 56, 58, 64  
 RawJSON, 153, 155  
 R2D2 code comment, 444  
 RealDigits, 60  
 Real numbers, 57  
 Rectified linear unit (ReLU), 369  
 ReducedVectors, 346  
 Relational operators, 30, 31  
 ReLU activation, 427  
 ResidualHistogram, 318  
 ResidualPlot, 318  
 Residuals, 266  
 ResourceData command, 278, 280  
 Resource Dataset, 421  
 ResourceObject, 275, 422  
 ResourceObject Fisher's Irises, 276  
 Retraining model hyperparameters  
     graphs and metrics, 320  
     plots, 320  
 Root mean squared error (RMSE), 317  
 Round function, 63

RoundLossList, 415  
 RoundMeasurementsLists, 415  
 r-squared value, 317

## S

Sector chart, 251  
 SectorChart command, 251  
 SeedRandom, 234  
 SemanticImport  
     comparison, 164  
     costume import, 164–166  
     CSV file, 157  
     datasets (*see* Datasets)  
     import data, 158  
     quantities, 158–160  
     Semantic objects, 158  
 SepalLength, 282, 283  
 SetPrecision, 60  
 SinglePredictionConfidenceInterval  
     Table option., 267  
 Six-digit precision, 60  
 Softmax function, 372  
 SoftmaxLayer, 372, 373  
 SoftPlus function, 370  
 Solve function, 35  
 Standard deviation, 241  
 Standard normal distribution, 249  
 Standard score, 241  
 Statistical Charts  
     bar graph, 242  
     Mathematica, 242  
     qualitative variable, 243  
 Statistical measures  
     data analysis, 239  
 StatsFun function, 124  
 StochasticGradientDescent method,  
     319, 328

- Stochastic gradient
  - descent (SGD), 412
  - Adam optimizer, 413
  - mathematical, 413
- Strings, 25–27
- SummationLayer, 388
- Syntax notations, 68
- System sampling, 237
  - elements, 237, 238
  - interval, 237
  - MapAt and Style, 238
  - non-random numbers, 239

## T

- TableRows, 290
- Tables
  - automated forms, 98
  - Background option, 100
  - contents, 96
  - dividers and spacers, 100
  - Grid, 99
  - headers, 99
  - labeling, 98
  - rows and columns, 97
  - TableForm, 96
  - Titles, 97
- Tab-separated value (TSV), 148–150, 275
- Tanh[x] function layer, 369
- Text, 25
- Text cells, 25
- Text formats, 25
- Text processing, 12, 13
- 3D graphics, 219
- 3D grid charts, 252
- 3D plot figure, 218
- 3D scatter plot, 228, 300, 301
- Titanic dataset, 321, 324, 327

- Tooltip, 200
  - curve expression, 199
  - plot expression, 199
- Tooltips, 199
- ToString command, 26
- Trained classifier function, 329
- TrainingProgressMeasurements, 430
- TrainingProgressReporting, 313
- Training set measurements, 433
- Transcendental numbers, 58
- Transpose function, 126
- Trigonometric functions, 61, 62
- 2D and 3D plots, 301
- 2D graphics, 219
- 2D plot theme, 229
- 2D scatter plot, 296, 300, 305
- Two-variable function, 222

## U

- UninitializedNet, 417
- UntrustedPath directories, 54
- User-defined functions, 93, 94, 196
- User interfaces (UIs), 9

## V

- Validation set measurements, 434
- Variables, 19, 21
- Vectors, 72, 73
- Violin diagram, 254
- Violin plots, 255, 256

## W

- Web data, 156, 157
- Wolfram Alpha, 37
  - algebraic equations, 37

## INDEX

### Wolfram Alpha (*cont.*)

- input code, 38, 39

- population of Australia, 39

- query, 37, 38, 40

- Tesla stock, 39

Wolfram Data Repository, 271, 275, 281,  
301, 419, 424

- HTTP response object, 272

- life Science category, 273

- Mathematica, 275

- website, 272

Wolfram Documentation Center, 50, 52

Wolfram ImageIdentify Net V1, 421

Wolfram Language, 2, 3, 6, 226, 269, 297,  
305, 327, 360, 361, 370, 373, 397  
and MXNet, 399

Wolfram Neural Net Repository, 418,  
425, 437

- AudioSet Data, 419

- neural network models, 418

Wolfram Neural Network, 362

Wolfram Prompt Repository, 441

## X, Y

Xavier method, 363

XLSX files

- CharacterEncoding option, 151

- grocery list dataset, 152

- import data, 150

- NaN-filled dataset, 152, 153

- SheetCount and Sheets, 151

- TableView command, 151, 152

XOR operator, 32

## Z

z-score, 241