

Introduction to programming for data science

STAT 201 Spring 2025

Lizhen Shi

Table of contents

Preface	11
I Getting started: Coding environment	12
1 Setting up your environment with VS Code	13
1.1 Learning Objectives	13
1.2 Introduction to Visual Studio Code (VS Code)	13
1.2.1 Core Features	13
1.2.2 User Interface	14
1.2.3 Extensions	15
1.2.4 Use Cases	16
1.2.5 Cross-Platform	16
1.3 Installing Visual Studio Code	16
1.4 Setting Up Python Development Environment in VS Code using python <code>venv</code> .	17
1.4.1 Install Python	17
1.4.2 Install Visual Studio Code Extensions	18
1.4.3 Set Up a Python Workspace for this course	18
1.4.4 Create a Notebook for your work	18
1.4.5 Create a Python environment for your work - GUI method	19
1.4.6 Choose the <code>.venv</code> environment as the kernel to run the notebook	20
1.4.7 Installing <code>ipykernel</code> for your notebook	21
1.5 Jupyter Notebooks in VS Code	21
1.5.1 Writing and executing code	21
1.5.2 Saving and loading notebooks	22
1.6 Rendering notebook as HTML using Quarto	23
1.6.1 Installing and Verifying Quarto	23
1.6.2 Converting the Notebook to HTML	24
1.7 In-class exercise	24
1.8 Reference	24
2 Setting File Paths and Understanding the Current Working Directory	25
2.1 Learning Objectives	25
2.2 What is the Current Working Directory?	25

2.3	Abosolute vs. Relative Paths	26
2.3.1	Absolute Path	26
2.3.2	Relative Path	26
2.3.3	Let's Try: Reading a Data File	26
2.3.4	Practice Exercise	28
2.4	Specifying Your File Path	29
2.4.1	Methods to Specify file path in Windows	29
2.5	Recognizing & Navigating to the Correct Working Directory in the Terminal (for generating html from <code>.ipynb</code> notebooks)	30
2.5.1	Step 1: What is the Working Directory in the Terminal?	30
2.5.2	Step 2: See What Files are in the current working directory	31
2.5.3	Step 3: Navigate to the Folder that Contains Your Notebook	31
2.5.4	Step 4: Generate the HTML file	31
2.5.5	Common Pitfall	31
2.5.6	Summary Cheat Sheet	31
2.6	Independent Study	32

II Python 33

3 Variables, expressions and statements 34

3.1	Commenting code	34
3.1.1	Practice exercise 1	34
3.2	<code>print()</code> function in python	34
3.2.1	Basic Examples	35
3.2.2	concatenating strings and variables/expressions in <code>print()</code> function . .	35
3.2.3	Customizing output formatting	37
3.2.4	Practice exercise 2	39
3.3	Data types	39
3.3.1	Primitive Data Types	39
3.3.2	Practice exercise 3	40
3.3.3	Commonly Used Built-in methods associated with each data type	40
3.4	Variables	47
3.4.1	Variable Declaration:	47
3.4.2	Dynamic Typing:	47
3.4.3	Variable Naming Rules:	48
3.4.4	Best Practices:	48
3.4.5	Checking Variable Types	48
3.4.6	Practice exercise 4	49
3.5	Assignment statements	49
3.6	Expressions	49
3.6.1	Mathematical Operations and Their Operators in Python	49
3.6.2	Operator Precedence in Python	50

3.6.3	Practice exercise 5	51
3.7	User input	51
3.7.1	Examples	52
3.7.2	Practice exercise 6	52
3.8	Converting data types in Python	53
3.8.1	Why Convert Data Types in Python?	53
3.8.2	How to Convert Data Types in Python	54
3.9	Errors and Exceptions	54
3.9.1	Syntax errors	54
3.9.2	Exceptions	55
3.9.3	Exception Handling	56
3.9.4	Practice exercise 7	57
3.9.5	Practice exercise 8	57
3.9.6	Semantic errors (bugs)	58
3.9.7	Practice exercise 9	58
4	Control flow tools	60
4.1	Indentation in Python	60
4.1.1	What is Indentation?	60
4.1.2	Rules for Indentation	60
4.1.3	Examples	61
4.2	Conditonal execution	61
4.2.1	Comparison operators	61
4.2.2	Logical Operators in Conditional Statements	62
4.2.3	if-elif-else statement	63
4.2.4	Practice exercise 1	64
4.2.5	Practice exercise 2	64
4.3	Loops in Python	66
4.3.1	Using range() in for Loops	66
4.3.2	while loop	71
4.3.3	Practice exercise 3	73
4.4	Control flow statements	75
4.4.1	break statement	75
4.4.2	Practice exercise 4	76
4.4.3	continue statement	76
4.4.4	Practice exercise 5:	77
4.4.5	pass statement	77
4.5	Loops with strings	77
4.5.1	Practice exercise 6	79
5	Functions	81
5.1	Function Definition	81
5.1.1	Why Use Functions?	81

5.2	Advantages of Functions	81
5.3	Types of Functions	82
5.3.1	Functions	82
5.3.2	Practice exercise 1	87
5.3.3	Practice exercise 2	87
5.4	User-defined Functions	88
5.4.1	Key Components (Based on the Diagram)	88
5.4.2	Functions are lazy	89
5.4.3	Arguments and Parameters in a Function	89
5.4.4	Type of Arguments in Python	90
5.4.5	Practice exercise 3	95
5.4.6	Functions that return objects	95
5.4.7	Practice exercise 4: Create a Custom Calculator	96
5.4.8	Bonus question: Calculator Function with Variable Number of Inputs	96
5.4.9	Practice exercise 5: Palindrome Checker	97
5.4.10	Global and local variables with respect to a function	97
5.4.11	Practice exercise 6	101
6	Data structures	102
6.1	Lists	102
6.1.1	Creating a List	102
6.1.2	Accessing Elements	103
6.1.3	Modifying a List	104
6.1.4	Practice exercise 1	109
6.1.5	List Comprehension	110
6.1.6	Practice exercise 2	112
6.2	Tuples	112
6.2.1	Creating a Tuple	112
6.2.2	Accessing Elements	112
6.2.3	Immutability	113
6.2.4	Tuple methods	113
6.2.5	Concatenating tuples	114
6.2.6	Why Use a Tuple?	114
6.2.7	Tuple Comprehension?	116
6.2.8	Practice exercise 3	117
6.2.9	Practice exercise 4	118
6.3	Sets	118
6.3.1	Creating a set	119
6.3.2	Accessing Elements	119
6.3.3	Adding and Removing Items	120
6.3.4	Mathematical Set Operations	121
6.3.5	Set Comprehension	122
6.3.6	Practice exercise 5	122

6.4	Dictionary	123
6.4.1	Creating a dictionary	123
6.4.2	Accessing and Modifying Values	124
6.4.3	Removing Keys	124
6.4.4	Iterating over elements of a dictionary	125
6.4.5	Practice exercise 6	125
6.4.6	Practice exercise 7	126
6.5	Choosing the Right Data Structure	127
6.6	Immutable and mutable Data Types in Python	128
6.7	Final Thoughts	130
6.7.1	Bonus Practice exercise	130
6.8	Resources	132
7	Python Iterables	133
7.1	What are Python Iterables	133
7.1.1	What Makes an Object Iterable?	133
7.1.2	Checking if an Object is Iterable	133
7.2	Common Iterables in Python	134
7.2.1	Strings	134
7.2.2	Lists	134
7.2.3	Tuples	134
7.2.4	Sets	135
7.2.5	Dictionaries	135
7.2.6	Ranges	135
7.3	Iterating over Iterables	135
7.4	Iterables Unpacking	136
7.4.1	Basic Unpacking	136
7.4.2	Extended (*) Unpacking	137
7.4.3	Unpacking with Functions	137
7.5	Built-in Functions for Iterables	138
7.5.1	General Functions	138
7.5.2	<code>sorted()</code>	139
7.5.3	<code>enumerate()</code>	141
7.5.4	<code>zip()</code>	143
7.5.5	unzipping	146
7.6	Practice exercises	146
7.6.1	Exercise 1: Iterating Using a For Loop and Manual Iteration	146
7.6.2	Exercise 2: Combining Iterables with <code>zip()</code>	146
7.6.3	Exercise 3: Unzipping a List of Tuples	147
7.6.4	Exercise 4: Using Built-in Functions on Iterables	147

8	Object-Oriented Programming	148
8.1	Classes	148
8.1.1	Built-in Data Types in Python	148
8.1.2	Understanding Classes in Python	149
8.1.3	Creating your own classes	149
8.2	Objects	151
8.2.1	Example: A class that analyzes a string	154
8.2.2	Practice exercise 1	155
8.3	Class Constructors	157
8.3.1	Default Constructor (No Parameters)	158
8.3.2	Using Default Values in Constructors	158
8.4	Difference Between Instance Attributes and Class Attributes in Python (OOP)	159
8.4.1	Instance Attributes (Defined in <code>__init__</code>)	159
8.4.2	Class Attributes (Defined Outside <code>__init__</code>)	160
8.5	Inheritance in Python	160
8.5.1	What is Inheritance?	160
8.5.2	Defining a Parent (Base) and Child (Derived) Class	161
8.5.3	The <code>super()</code> Function	162
8.5.4	Method Overriding in Inheritance	162
8.5.5	Check Relationship	163
8.5.6	The <code>object</code> class in Python	164
8.5.7	Practice exercise 2	165
8.5.8	Practice exercise 3	166
III	R	169
9	R: Basics	170
9.1	Comments	170
9.2	Data Types	171
9.2.1	Numeric Data	171
9.2.2	Character Data	171
9.2.3	Logical Data	172
9.3	Variables	173
9.3.1	The assignment operator	173
9.4	Converting datatypes	174
9.5	<code>typeof()</code> vs. <code>class()</code> in R	176
9.5.1	Key Differences:	177
9.5.2	Common Data Types	177
9.6	Displaying information	178
9.6.1	Using <code>print()</code>	178
9.6.2	Using <code>cat()</code>	178
9.6.3	Using <code>paste()</code>	178

9.6.4	Using <code>message()</code>	178
9.6.5	Using <code>sprintf()</code>	179
9.7	Taking user input	179
9.8	Arithmetic Operations	180
9.9	Comparison Operators	180
9.10	Logical Operators	181
9.11	2. Differences Between Element-wise and Short-circuit Operators	182
10	R: Control flow tools	184
10.1	Code Blocks: R vs. Python	184
10.1.1	Code blocks in R (Uses <code>{}</code>)	184
10.2	TryCatch	185
10.3	Control flow tools	186
10.4	Conditional statements	187
10.5	Loops	187
10.5.1	The <code>:</code> operator	187
10.5.2	<code>seq()</code> function	189
10.5.3	<code>for</code> loop	189
10.5.4	<code>while</code> loop	191
10.6	Loop control statements	192
10.6.1	<code>break</code> statement (Loop Termination)	192
10.6.2	<code>next</code> statement (Skip Iteration, Similar to <code>continue</code> in Python)	193
10.6.3	Practice exercise	194
10.7	characters in R	194
10.7.1	Basic Definitions	194
10.7.2	String/Character Length	195
10.7.3	String Indexing and Slicing	195
10.7.4	String Concatenation	195
10.7.5	Changing Case (Uppercase & Lowercase)	196
10.7.6	Finding Substrings	196
10.7.7	String Replacement	196
10.7.8	String Splitting	197
10.7.9	Summary table	197
10.8	Practice exercises	198
10.8.1	Practice exercise 1	198
10.8.2	Practice exercise 2	198
10.8.3	Practice exercise 3	198
10.8.4	Practice exercise 4	199
10.8.5	Practice exercise 5	200
11	R: Functions	201
11.1	Functions in R	201
11.1.1	Built-in Functions in R	201

11.2	User-defined Functions	206
11.2.1	Defining a function	206
11.2.2	Calling a function	206
11.2.3	Arguments	207
11.2.4	Return values	207
11.3	Function arguments	208
11.3.1	Positional arguments	208
11.3.2	default arguments	209
11.3.3	keyword arguments	209
11.3.4	Variable-number arguments	210
11.4	Variable scope	211
11.4.1	Local variables	211
11.4.2	Global variables	212
11.4.3	Modifying Global variables inside a function	212
11.5	Practice exercises	213
11.5.1	Problem 1	213
11.5.2	Problem 2	213
11.5.3	Problem 3	213
12	R: Atomic vectors	214
12.0.1	Creating a vector	214
12.0.2	Atomic vector types	215
12.0.3	Slicing a vector	217
12.0.4	Mutating a vector	218
12.0.5	Removing elements from a vector	219
12.0.6	Iterating	219
12.0.7	Using as a function input	219
12.0.8	Applying a Function to Each Element of a Vector	220
12.0.9	The <code>rep()</code> function	221
12.0.10	The <code>which()</code> function	221
12.0.11	Element-wise operations on atomic vectors	222
12.0.12	Commonly Used Functions for Atomic Vectors	224
12.1	Vectorization in R	225
12.1.1	Comparing Loop vs. Vectorized Operations	225
12.1.2	Performance Comparison: Loop vs. <code>sapply()</code> vs. Vectorized	226
12.1.3	Functions that support vectorization	227
12.1.4	Understanding why vectorization is faster	228
12.2	Practice exercises	228
12.2.1	Exercise 1	228
12.2.2	Exercise 2	229
12.2.3	Exercise 3	230

13 R: Data structures	232
13.1 Atomic vectors ((1D, Homogeneous))	233
13.2 Matrix ((2D, Homogeneous))	233
13.2.1 Creating a matrix	233
13.2.2 Getting matrix dimensions, the number of rows, the number of columns	234
13.2.3 Matrix indexing and Slicing	234
13.2.4 Working with <code>rownames()</code> and <code>colnames()</code>	235
13.2.5 Adding and Removing	236
13.2.6 Iterating	238
13.2.7 Element-wise arithmetic operations	238
13.2.8 The <code>apply()</code> function	240
13.2.9 Misc useful functions:	242
13.3 changes	243
13.4 Lists (1D, Heterogeneous)	243
13.4.1 Creating and Indexing	243
13.4.2 Naming a list	244
13.4.3 Using as Function Input	244
13.4.4 Applying a function to each element of a list: the <code>lapply()</code> function . .	245
13.5 Data Frames (2D, Heterogeneous)	246
13.5.1 Creating data Frames	247
13.5.2 Common functions	247
13.5.3 Missing values	247
13.5.4 Accessing Data Frame Elements	249
13.5.5 Using Logical Operators for Data Frame Subsetting	249
13.5.6 Iterating	251
13.6 Summary	252
13.7 Practice exercises	252
13.7.1 Exercise 1	252
13.7.2 Exercise 2	256
13.7.3 Exercise 3	258
 Appendices	 260
A Assignment templates and Datasets	260

Preface

This book serves as the course textbook for **STAT 201, Winter 2025** at Northwestern University. It is an evolving resource designed to support the course's learning objectives. This edition builds upon the foundational work of **Professor Arvind Krishna**, whose contributions have provided a strong framework for this resource. We are deeply grateful for his efforts, which continue to shape the book's development.

Throughout the quarter, the content will be updated and refined in real time to enhance clarity, depth, and relevance. These modifications ensure alignment with current teaching objectives and methodologies.

As a **living document**, this book welcomes feedback, suggestions, and contributions from students, instructors, and the broader academic community. Your input helps improve its quality and effectiveness.

Thank you for being part of this journey—we hope this resource serves as a valuable guide in your learning.

Part I

Getting started: Coding environment

1 Setting up your environment with VS Code

`<IPython.core.display.Image object>`

1.1 Learning Objectives

By completing this lecture, you will be able to:

- Install and configure Visual Studio Code (VS Code) for Python programming.
- Leverage Jupyter Notebook within VS Code for your data science Python programming.
- Use Quarto to create HTML documents for your upcoming homework submissions.

1.2 Introduction to Visual Studio Code (VS Code)

Visual Studio Code (VS Code) is a free, open-source, and lightweight code editor developed by Microsoft. It's widely used for coding, debugging, and working with **various programming languages** and frameworks. Here's an overview of its key features and functionalities:

1.2.1 Core Features

- **Multi-language Support:** VS Code supports a wide range of programming languages out of the box, including Python, JavaScript, TypeScript, HTML, CSS, and more. Additional language support can be added via extensions.
- **Extensibility:** The editor has a rich ecosystem of extensions available through the Visual Studio Code Marketplace. These extensions add support for additional programming languages, themes, debuggers, and tools like Git integration.
- **IntelliSense:** Provides intelligent code completion, parameter info, quick info, and code navigation for many languages, enhancing productivity and reducing errors.
- **Integrated Terminal:** Allows you to run command-line tools directly from the editor, making it easy to execute scripts, install packages, and more without leaving the coding environment.

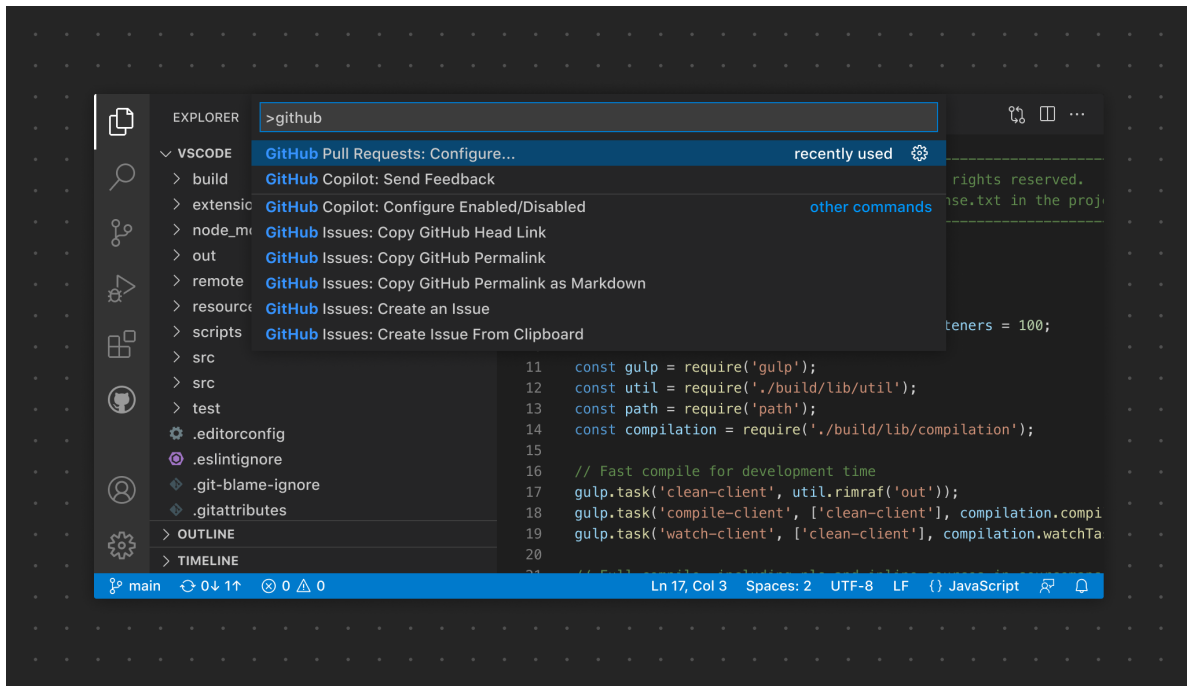
- **Version Control Integration:** Seamless integration with Git and other version control systems, allowing you to manage source code repositories, stage changes, commit, and view diffs within the editor.
- **Debugging:** Supports debugging with breakpoints, call stacks, and an interactive console for various languages and frameworks.

1.2.2 User Interface

- **Editor:** The main area to edit your files. You can open as many editors as you like side by side vertically and horizontally.
- **Primary Side Bar:** Contains different views like the Explorer to assist you while working on your project.
- **Activity Bar:** Located on the far left-hand side. Lets you switch between views and gives you additional context-specific indicators, like the number of outgoing changes when Git is enabled. You can change the position of the Activity Bar.
- **Panel:** An additional space for views below the editor region. By default, it contains output, debug information, errors and warnings, and an integrated terminal. The Panel can also be moved to the left or right for more vertical space.



- **Command Palette:** Accessed with `Ctrl+Shift+P` (or `Cmd+Shift+P` on macOS), it provides a quick way to execute commands, switch themes, change settings, and more.



1.2.3 Extensions

- **Language Extensions:** Add support for additional languages such as Rust, Go, C++, and more.
- **Linters and Formatters:** Extensions like ESLint, Prettier, and Pylint help with code quality and formatting.
- **Development Tools:** Extensions for Docker, Kubernetes, database management, and more.
- **Productivity Tools:** Extensions for snippets, file explorers, and workflow enhancements.



1.2.4 Use Cases

- **Web Development:** VS Code is popular among web developers for its robust support for HTML, CSS, JavaScript, and front-end frameworks like React, Angular, and Vue.
- **Python Development:** With the Python extension, it provides features like IntelliSense, debugging, linting, and Jupyter Notebook support.
- **Data Science:** Supports Jupyter notebooks, allowing data scientists to write and run Python code interactively.
- **DevOps and Scripting:** Useful for writing and debugging scripts in languages like PowerShell, Bash, and YAML for CI/CD pipelines.

1.2.5 Cross-Platform

- Available on Windows, macOS, and Linux, making it accessible to developers across different operating systems.

Overall, VS Code is a versatile and powerful tool for a wide range of development activities, from simple scripting to complex software projects.

1.3 Installing Visual Studio Code

- Step 1: **Download VS Code:**

- Go to the [official VS Code website](#) and download the installer for your operating system.
- **Step 2: Install VS Code:**
 - Run the installer and follow the prompts to complete the installation.
- **Step 3: Launch VS Code:**
 - Open VS Code after installation to ensure it's working correctly.

1.4 Setting Up Python Development Environment in VS Code using python venv

Unlike Spyder and PyCharm, which are specifically designed for Python development, VS Code is a versatile code editor with multi-language support. As a result, setting up the Python environment requires some additional configuration.

This step-by-step guide will walk you through setting up your Python environment in Visual Studio Code from scratch using `venv`.

1.4.1 Install Python

For this course, any version of Python 3 works. You don't need to worry about having the latest version of Python, as long as you have Python 3 installed.

If Python 3 is already installed on your computer, you can skip this step.

1. Download Python:

- Go to the [official Python website](#) and download the latest version of Python for your operating system.
- Ensure that you check the box “**Add Python to PATH**” during installation if it exists.

2. Verify Python Installation:

- Open a terminal (Command Prompt on Windows, Terminal on macOS/Linux) and type:

```
python --version
```

- You should see the installed Python version. If the command line doesn't work, you might see an error message like:
 - `python is not recognized`

– `python` command is not found

This issue is often caused by Python not being added to the **PATH** environment variable. Please refer to [the instructions](#) to resolve this issue.

Note:

Before moving forward, ensure that the command `python --version` successfully prints the version of your installed Python. If it does not, you may need to troubleshoot your Python installation or add it to the PATH environment variable.

1.4.2 Install Visual Studio Code Extensions

1. **Open VS Code.**

2. **Go to Extensions:**

- Click on the Extensions icon on the sidebar or press `Ctrl+Shift+X`.

3. **Install Python Extension:**

- Search for the “Python” extension by Microsoft and install it.

4. **Install Jupyter Extension:**

- Search for the “Jupyter” extension by Microsoft and install it.

1.4.3 Set Up a Python Workspace for this course

1. **Create a New Folder:**

- Create a new folder on your computer where you want to store your Python code for this course.

2. **Open Folder in VS Code:**

- Go to `File > Open Folder` and select the newly created folder.

1.4.4 Create a Notebook for your work

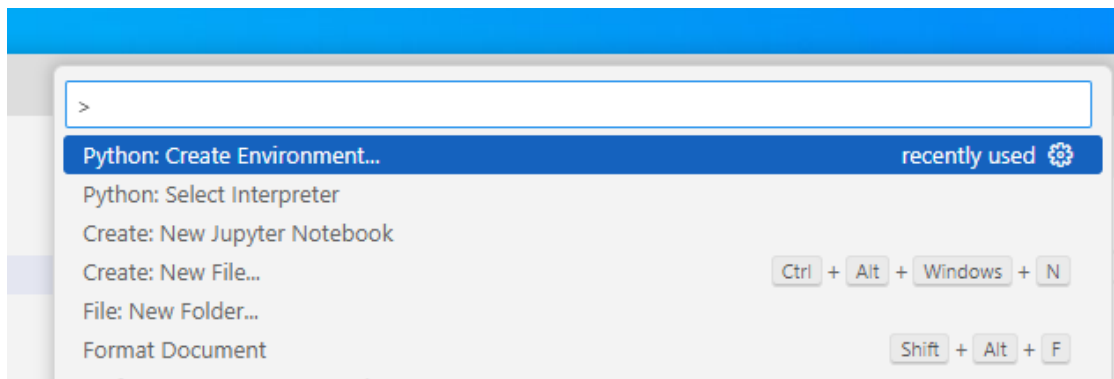
- In VS Code, go to `File > New File` and select **Jupyter Notebook**. You should see a blank notebook named `Untitled-1.ipynb` as in the figure below. The `.ipynb` extension stands for IPython Notebook.



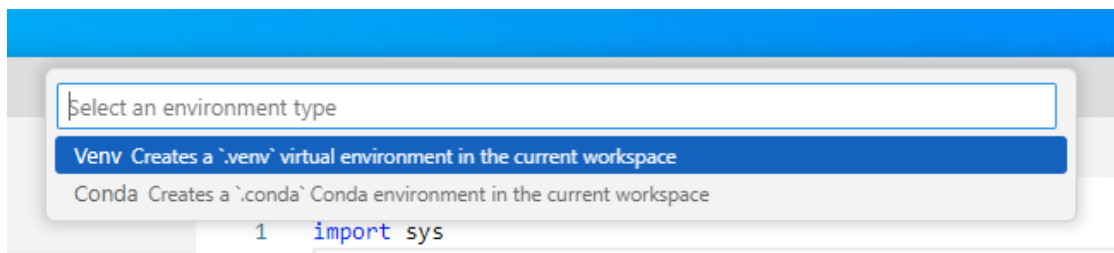
Jupyter Notebook is an **interactive platform** that allows you to write code, add text, and create visualizations. Data scientists love using Jupyter Notebooks as an alternative to working directly with Python files because of their interactivity and flexibility.

1.4.5 Create a Python environment for your work - GUI method

- When you start a Jupyter Notebook in VS Code, you need to choose a kernel. Kernel is the “engine” that runs your code within your Jupyter notebook, and it is tied to a specific Python interpreter or environment.
 - **What’s the difference between an interpreter and an environment?** An interpreter is a program that runs your Python code. An environment, on the other hand, is a standalone “space” where your code runs. It’s like a container that holds its own interpreter and environment-specific libraries/dependencies, so each project can have its own environment setup without affecting others.
 - **Why do we prefer creating an environment for this course rather than using the global interpreter that comes with your Python installation?** As a data scientist, you may work on multiple projects and attend different courses that require different sets of packages, dependencies, or even Python versions. By creating a separate environment, you can prevent conflicts between libraries, dependencies, and Python versions across your projects ([dependency hell](#)) and also ensure code reproducibility. It is always good practice to work within python environments, especially when you have different projects going on.
 - Let’s create a Python environment for the upcoming coursework.



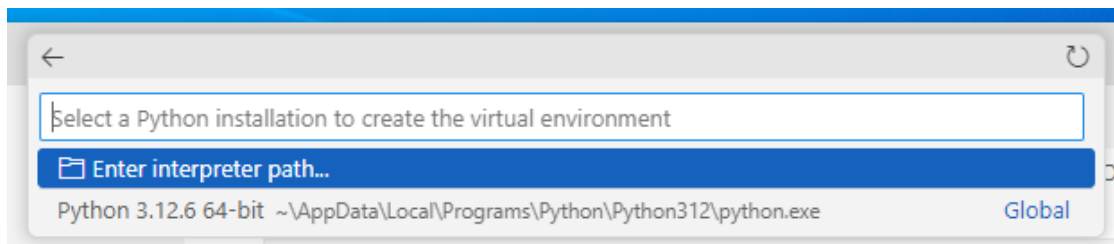
- Create using **venv** in the current workspace



What is venv

In Python, **venv** stands for Virtual Environment, which is a tool used to create isolated environments for Python projects. This helps manage dependencies and avoid conflicts between different projects that may require different versions of Python or different packages.

- Choose a specific python interpreter for your environment:



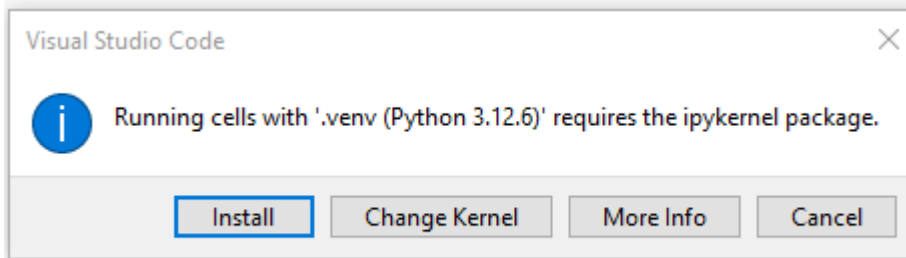
Congratulations! A virtual environment named **.venv** has been successfully created in your project folder.

1.4.6 Choose the **.venv** environment as the kernel to run the notebook

For all your upcoming work in this project, you can select this environment to ensure a consistent setup.

1.4.7 Installing ipykernel for your notebook

Create a code cell in the notebook and run it. The first time you run a code cell, you will run into



- After installing ipykernel, you should be able to run the following cell.

```
import sys
print("Current Python executable:", sys.executable)
```

Current Python executable: c:\Users\lsi8012\OneDrive - Northwestern University\FA24\303-1\te

`sys.executable` is an attribute in the Python `sys` module that returns the path to the Python interpreter that is currently executing your code.

1.5 Jupyter Notebooks in VS Code

After setting up your environment and successfully running your notebook using the created environment, follow this [instruction](#) to become familiar with the native support for Jupyter Notebooks in VS Code

1.5.1 Writing and executing code

Code cell: By default, a cell is of type *Code*, i.e., for typing code, as seen as the default choice in the dropdown menu below the *Widgets* tab. Try typing a line of python code (say, `2+3`) in an empty code cell and execute it by pressing *Shift+Enter*. This should execute the code, and create a new code cell. Pressing *Ctrl+Enter for Windows (or Cmd+Enter for Mac)* will execute the code without creating a new cell.

Commenting code in a code cell: Comments should be made while writing the code to explain the purpose of the code or a brief explanation of the tasks being performed by the

code. A comment can be added in a code cell by preceding it with a `#` sign. For example, see the comment in the code below.

Writing comments will help other users understand your code. It is also useful for the coder to keep track of the tasks being performed by their code.

```
#This code adds 3 and 5
3+5
```

8

Please refer to the [Style Guide for Python Code](#) to develop good coding habits from the start.

Markdown cell: Although a comment can be written in a code cell, a code cell cannot be used for writing headings/sub-headings, and is not appropriate for writing lengthy chunks of text. In such cases, change the cell type to *Markdown* from the dropdown menu below the *Widgets* tab. Use any markdown cheat sheet found online, for example, [this one](#) to format text in the markdown cells.

Jupyter Notebook shortcuts: Jupyter Notebook shortcuts are handy and quick. Here is a list of commonly used shortcuts for beginners:

- **Shift + Enter:** Run the cell and move to the next one.
- **A:** Insert a cell above.
- **B:** Insert a cell below.
- **D, D:** Delete the selected cell.
- **Y:** Change the cell to Code type.
- **M:** Change the cell to Markdown type.

1.5.2 Saving and loading notebooks

To save the notebook in VS Code, click on **File** and select **Save As**, or use the keyboard shortcut **CTRL+S**. Your notebook will be saved as a file with the extension `.ipynb`. This file will contain all the code and outputs and can be opened and edited with VS Code or any Jupyter-compatible environment.

To open an existing Jupyter notebook with VS Code: 1. Navigate to the file in your system's file explorer. 2. Right-click on the file. 3. Select **Open with VS Code**.

1.6 Rendering notebook as HTML using Quarto

Quarto is designed for high-quality, customizable, and publishable outputs, making it suitable for reports, blogs, or presentations. We'll use Quarto to print the `**.ipynb*` file as HTML for your assignment submission.

1.6.1 Installing and Verifying Quarto

- Download and install Quarto from the [official website](#).
- Follow the installation instructions for your operating system.
- Open your terminal within VS Code:
 - Go to **Terminal** -> **New Terminal**.
- Run the following command to verify that Quarto and its dependencies are correctly installed: `quarto --version`
- You should see the installed quarto version. If the command line doesn't work, you might see an error message like:
 - ``quarto is not recognized``
 - ``quarto command is not found``

This issue is often caused by Quarto not being added to the PATH environment variable. Similar to how you added the Python path to the environment variable above, you need to add the Quarto path to the system environment variable so that the command can be recognized by your operating system's shell.

On Windows, if you used the default installation path (without changing it), Quarto is installed in: `C:\Users\<USER>\AppData\Local\Programs\Quarto\bin`

Note:

Before moving forward, ensure that the command `quarto --version` successfully prints the version of your installed quarto. If it does not, you may need to troubleshoot your quarto installation or add it to the PATH environment variable.

1.6.2 Converting the Notebook to HTML

Check the procedure for rendering a notebook as HTML [here](#). You have several options to format the file. Here are some points to remember when using Quarto to render your notebook as HTML:

1. The [Raw NBConvert](#) cell type is used to render different code formats into HTML or LaTeX. This information is stored in the notebook metadata and converted appropriately. **Use this cell type to put the desired formatting settings for the HTML file.**
2. In the formatting settings, remember to use the setting `embed-resources: true`. This will ensure that the rendered HTML file is self-contained, and is not dependent on other files. This is especially important when you are sending the HTML file to someone, or uploading it somewhere. If the file is self-contained, then you can send the file by itself without having to attach the dependent files with it.

Once you have entered the desired formatting setting in the `Raw NBConvert` cell, you are ready to render the notebook to HTML. Open the terminal, navigate to the directory containing the notebook (*.ipynb file*), and use the command: `quarto render filename.ipynb --to html`.

1.7 In-class exercise

1. Create a new notebook.
2. Save the file as `In_class_exercise_1`.
3. Give a heading to the file - `First HTML file`.
4. Print `Today is day 1 of my programming course`.
5. Compute and print the number of seconds in a day.
6. Generate html from the notebook using Quarto

The HTML file should look like the picture below.

```
<IPython.core.display.Image object>
```

1.8 Reference

- [Getting Started with VS Code](#)
- [Jupyter Notebooks in VS Code](#)
- [Quarto](#)

2 Setting File Paths and Understanding the Current Working Directory

<IPython.core.display.Image object>

2.1 Learning Objectives

By the end of this lesson, students should be able to:

- Check Your current working directory in Python.
- Use relative and absolute paths correctly.
- Read files using proper path specification.
- Fix the common file path errors in VS Code.
- Generate HTML from your notebook

2.2 What is the Current Working Directory?

- In Python, the current working directory (CWD) is the folder your code is running from.
- When you use a relative path, it's relative to this directory.

In python, the `os` module allow you to interact with the OS and the filesystem. We need to import it before using it

```
import os  
  
print(os.getcwd())
```

```
c:\Users\lsi8012\workspace\Intro_to_programming_for_data_sci_wi25
```

This tells you where Python is currently looking for files by default.

2.3 Absolute vs. Relative Paths

2.3.1 Absolute Path

- Full path from the root of the file system.

```
path = "/Users/lizhen/Desktop/data/myfile.csv"
```

2.3.2 Relative Path

- Path relative to the current working directory.

```
path = "data/myfile.csv" # if your CWD is the parent folder of "data"
```

In your code, it's highly recommended using relative paths whenever possible for better portability!

2.3.3 Let's Try: Reading a Data File

Suppose your directory looks like this:

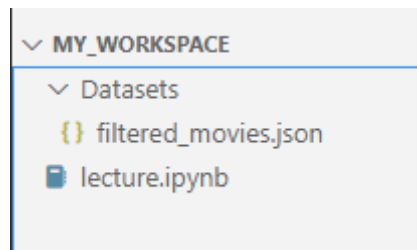


Figure 2.1: file_directory

In `lecture.ipynb`, you can do:

```
import json
with open("Datasets/filtered_movies.json", encoding="utf8") as file:
    movie_data=json.load(file)

movie_data[:2]
```

```
[{'Title': '12 Rounds',
  'US Gross': 12234694,
  'Worldwide Gross': 18184083,
  'US DVD Sales': 8283859,
  'Production Budget': 20000000,
  'Release Date': 'Mar 27 2009',
  'MPAA Rating': 'PG-13',
  'Running Time min': 108,
  'Distributor': '20th Century Fox',
  'Source': 'Original Screenplay',
  'Major Genre': 'Action',
  'Creative Type': 'Contemporary Fiction',
  'Director': 'Renny Harlin',
  'Rotten Tomatoes Rating': 28,
  'IMDB Rating': 5.4,
  'IMDB Votes': 8914},
{'Title': '2012',
  'US Gross': 166112167,
  'Worldwide Gross': 766812167,
  'US DVD Sales': 50736023,
  'Production Budget': 200000000,
  'Release Date': 'Nov 13 2009',
  'MPAA Rating': 'PG-13',
  'Running Time min': 158,
  'Distributor': 'Sony Pictures',
  'Source': 'Original Screenplay',
  'Major Genre': 'Action',
  'Creative Type': 'Science Fiction',
  'Director': 'Roland Emmerich',
  'Rotten Tomatoes Rating': 39,
  'IMDB Rating': 6.2,
  'IMDB Votes': 396}]
```

If you run into a `FileNotFoundError`, it means your file path is not specified correctly and the program is unable to locate the file.

```

-----
FileNotFoundError                                Traceback (most recent call last)
Cell In[17], line 2
      1 import json
----> 2 with open("filtered_movies.json", encoding="utf8") as file:
      3     movie_data=json.load(file)
      4 movie_data[:2]

File ~\AppData\Roaming\Python\Python312\site-packages\IPython\core\interactiveshell.
  317 if file in {0, 1, 2}:
  318     raise ValueError(
  319         f"IPython won't let you open fd={file} by default "
  320         "as it is likely to crash IPython. If you know what you are doing, "
  321         "you can use builtins' open."
  322     )
--> 324 return io_open(file, *args, **kwargs)

FileNotFoundError: [Errno 2] No such file or directory: 'filtered_movies.json'

```

2.3.4 Practice Exercise

What if your file directory look like this:

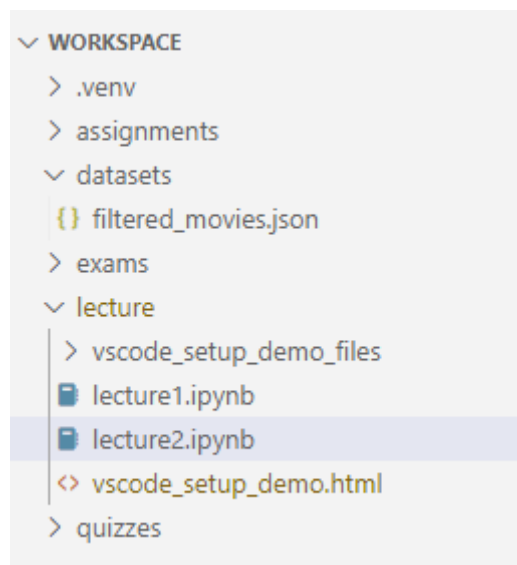


Figure 2.2: file_directory

You are now working in `lecture2.ipynb`. How should you specify the file path so that the

data in the JSON file can be read properly into your program?

2.4 Specifying Your File Path

We will discuss how to specify file paths in Python for loading and saving data, with an emphasis on **absolute** and **relative** paths. File paths are crucial when working with files in Python, such as reading datasets and writing results to files.

2.4.1 Methods to Specify file path in Windows

Windows uses backslashes (\) to separate directories in file paths. However, in Python, the backslash is an escape character, so you need to either: * Escape backslashes by using a double backslash (\\), or * Use raw strings by prefixing the path with r.

2.4.1.1 Method 1: Using escaped backslashes

```
file_path = "C:\\Users\\Username\\Documents\\data.csv"
```

2.4.1.2 Method 2: Using Raw Strings

```
file_path = r"C:\Users\Username\Documents\data.csv"
```

2.4.1.3 Method 3: Using forward slashes (/)

```
file_path = "C:/Users/Username/Documents/data.csv"
```

2.4.1.4 Method 4: Using os.path.join

```
import os
file_path = os.path.join("C:", "Users", "Username", "Documents", "data.csv")
```

This method works across different operating systems because `os.path.join` automatically uses the correct path separator (`\` for Windows and `/` for Linux/Mac).

macOS does not have the same issue as Windows when specifying file paths because macOS (like Linux) uses forward slashes (`/`) as the path separator, which is consistent with Python's expectations.

2.4.1.5 Best Practices for File Paths in Data Science

- Use relative paths when working in a project structure, as it allows the project to be portable.
- Use absolute paths when working with external or shared files that aren't part of your project.
- Check the current working directory to ensure you are referencing files correctly.
- Avoid hardcoding file paths directly in the code to make your code reusable on different machines.
- Use the forward slash (`/`) as a path separator or `os.path.join` to specify file path if your code will work across different operating systems

2.5 Recognizing & Navigating to the Correct Working Directory in the Terminal (for generating html from .ipynb notebooks)

Before exporting a notebook to `.html`, you must make sure your terminal is in the folder that contains your `.ipynb` file.

2.5.1 Step 1: What is the Working Directory in the Terminal?

The working directory is where your terminal is currently located in your computer's file system.

Run the following command in the terminal to print it out.

```
pwd
```

This tells you where your terminal is currently pointed.

2.5.2 Step 2: See What Files are in the current working directory

Run

```
ls
```

This will list the files/folders. Look for your `.ipynb` notebook.

If you see your notebook file here, you're in the right directory!

If not, you need to navigate to the correct folder.

2.5.3 Step 3: Navigate to the Folder that Contains Your Notebook

Run

```
cd <folder-name>
```

- Use `cd ..` to go up one folder
- Use `ls` after `cd` to check that you're in the right place

2.5.4 Step 4: Generate the HTML file

Once you are in the correct folder (containing your `.ipynb` file), run:

```
quarto render analysis.ipynb --to html
```

This will generate `analysis.html` in the same folder. Please replace the `analysis.ipynb` with your notebook name

2.5.5 Common Pitfall

Trying to export the notebook while the terminal is still in the wrong folder (like your home directory) will give you this error:

```
[NbConvertApp] WARNING | Notebook file 'analysis.ipynb' not found
```

Fix: Use `cd` to go to the correct directory first!

2.5.6 Summary Cheat Sheet

Command	Purpose
<code>pwd</code>	Show your current directory
<code>ls</code>	List files in the current folder
<code>cd folder/</code>	Enter a folder
<code>cd ..</code>	Go up one level
<code>quarto render file.ipynb --to html</code>	Convert to HTML

2.6 Independent Study

To reinforce the skills learned in this lecture, complete the following tasks:

1. Set Up Your Workspace

- Create a folder named **STAT201** for putting the course materials.
- Create a python environment for the upcoming coursework.
- Organize your files by creating separate directories for **datasets**, **assignments**, **quizzes**, **lectures**, and **exams**.
- Set up these directories in your file system to keep your work structured and easy to navigate.

2. Create your first notebook for quiz and generate html for your submission

1. Create a new notebook named `quiz1.ipynb` inside the `lecture` folder.
2. Add a heading to the top of the notebook: `# First HTML file`
3. In a code cell, print the message: `Today is day 1 of my programming course.`
4. In another code cell, compute and print the number of seconds in a day (i.e., $24 \times 60 \times 60$).
5. Use **Quarto** to convert the notebook to an HTML file.

The HTML file should look like the picture below.

```
<IPython.core.display.Image object>
```

By completing these exercises, you'll gain practical experience with file paths, generating HTML from notebooks using Quarto, and interacting with the filesystem in Jupyter notebooks. This will prepare you for Python programming starting next week.

Part II

Python

3 Variables, expressions and statements

<IPython.core.display.Image object>

3.1 Commenting code

The `#` symbol can be used to comment the code. Anything after the `#` sign is ignored by python. Commenting a code may have several purposes, such as:

- Describe what is going to happen in a sequence of code
- Document who wrote the code or other ancillary information
- Turn off a line of code - perhaps temporarily

For example, below is code with a comment to describe the purpose of the code:

```
#Computing number of hours of lecture in this course  
print("Total lecture hours of STAT201=",10*3*(5/6))
```

Total lecture hours of STAT201= 25.0

3.1.1 Practice exercise 1

Which of the following lines is a comment:

1. `#this is a comment`
2. `##this may be a comment`
3. `A comment#`

3.2 `print()` function in python

The `print()` function is a fundamental tool for displaying information.

3.2.1 Basic Examples

```
# Printing a simple string
print("Hello, World!")
```

Hello, World!

```
# Printing a string with a number
print ("The total number of seconds in a day is", 24*60*60)
```

The total number of seconds in a day is 86400

```
# combine multiple strings using the + operator
print("Hello, " + "World!")
```

Hello, World!

3.2.2 concatenating strings and variables/expressions in print() function

3.2.2.1 Using f-Strings (Formatted String Literals)

f-strings provide a concise way to embed expressions inside strings. Introduced in Python 3.6, they improve readability and efficiency.

3.2.2.2 Syntax

- Use **f** or **F** before the string.
- Embed variables or expressions in **{}**.

```
# Example 1: Basic Variable Substitution
name = "Alice"
age = 30
print(f"My name is {name} and I am {age} years old.")
```

```
# Example 2: Arithmetic Expressions
a = 10
b = 5
```

```
print(f"The sum of {a} and {b} is {a + b}.")

# Example 3: Formatting Numbers
pi = 3.14159
print(f"Pi rounded to 2 decimal places is {pi:.2f}.")
print(f"Pi rounded to 2 decimal places is {pi:.0f}.")
```

My name is Alice and I am 30 years old.
The sum of 10 and 5 is 15.
Pi rounded to 2 decimal places is 3.14.
Pi rounded to 2 decimal places is 3.

```
value = 123456789
print(f"Rounded With commas: ${value:,}")
```

Rounded With commas: \$123,456,789

3.2.2.3 Using str.format() Method

The `str.format()` method allows you to format strings by placing placeholders `{}` in the string and replacing them with variables or values.

"Your text here {}".format(variable_or_expression)

```
# Example 1: Basic Variable Substitution
name = "Bob"
age = 25
print("My name is {} and I am {} years old.".format(name, age))

# Example 2: Using Positional Arguments
print("The sum of {0} and {1} is {2}.".format(a, b, a + b))

# Example 3: Using Keyword Arguments
print("Pi rounded to 2 decimal places is {value:.2f}.".format(value=pi))
print("Pi rounded to 2 decimal places is {value:.0f}.".format(value=pi))
```

My name is Bob and I am 25 years old.
The sum of 10 and 5 is 15.
Pi rounded to 2 decimal places is 3.14.
Pi rounded to 2 decimal places is 3.

3.2.3 Customizing output formatting

The `print()` function in Python is highly customizable. By default, it adds a newline character (`\n`) at the end of each output and separates multiple arguments with a space (). However, these default behaviors can be changed using the `end` and `sep` parameters.

3.2.3.1 Default Behavior of `print()`

When you call `print()` multiple times, each statement starts on a new line:

```
print("Hello")  
print("World") # this is printed on a new line
```

Hello
World

3.2.3.2 Changing the `end` Parameter

To avoid automatic line breaks and control what is appended at the end of the output, use the `end` parameter.

```
print("Hello", end=" ")  
print("World") # this is printed on the same line
```

Hello World

```
print("Loading", end="...")  
print("Complete")
```

Loading...Complete

```
print("Line 1", end="")  
print("Line 2")
```

Line 1Line 2

3.2.3.3 Changing the sep Parameter

When printing multiple arguments, the default separator between them is a space. You can change this behavior using the `sep` parameter.

```
print("apple", "banana", "cherry", sep=", ")  
# Output: apple, banana, cherry
```

apple, banana, cherry

```
print("A", "B", "C", sep="")
```

ABC

```
print("python", "java", "c++", sep="\n")
```

python
java
c++

3.2.3.4 Combining end and sep

Both `end` and `sep` can be used together for more control:

```
print("1", "2", "3", sep="-", end=".")  
print(" Done!")
```

1-2-3. Done!

3.2.3.5 Takeaway:

- The `end` parameter changes what is appended at the end of the output (default: `\n`).
- The `sep` parameter modifies how multiple arguments are separated (default: space).
- Both can be combined to create customized output formatting.

3.2.4 Practice exercise 2

Use the `print()` function to:

- Display your name, age, and favorite hobby.
- Format the output neatly using f-strings.
- Use `sep` and `end` to produce this output: `apple:banana:cherry`.

3.3 Data types

Python provides several built-in data types for storing different kinds of information in variables. These data types can be broadly categorized into primitive data types and collection (containers) data types as shown at the beginning of this chapter. While collection data types will be covered in Chapter 5, this chapter focuses on primitive data types, which are used to represent a single value.

3.3.1 Primitive Data Types

They represent a single value. In Python, primitive data types include:

- **Integer (`int`)**: Whole numbers (e.g., 10, -3).
- **Floating-point number (`float`)**: Numbers with decimals (e.g., 3.14, -2.7).
- **Boolean (`bool`)**: Logical values `True` or `False`.
- **None type (`None`)**: Represents the absence of a value.
- **String (`str`)**: A sequence of characters (e.g., "hello", 'world').

The data type of the object can be identified using the in-built python function `type()`. For example, see the following objects and their types:

```
type(4)
```

```
int
```

```
type(4.4)
```

```
float
```

```
type('4')
```

str

```
type(True)
```

bool

3.3.2 Practice exercise 3

What is the datatype of the following objects?

1. 'This is False'
2. "This is a number"
3. 1000
4. 65.65
5. False

3.3.3 Commonly Used Built-in methods associated with each data type

3.3.3.1 Strings

Strings are sequences of characters and are immutable in Python.

Below are Commonly used Methods for strings:

- `lower()`: returns a string with every letter of the original in lowercase
- `upper()`: returns a string with every letter of the original in uppercase
- `replace(x,y)`: returns a string with every occurrence of x replaced by y
- `count(x)`: counts the number of occurrences of x in the string
- `index(x)`: returns the location of the first occurrence of x
- `format()`: format strings
- `isalpha()`: returns True if every character of the string is a letter

For a more comprehensive list of string methods, please refer to [here](#)


```

# Example Methods:
s = "Hello, World!"

# Returns the length of the string
len(s)
print("the length of the string is", len(s))

# Converts string to uppercase
s.upper()
print("the string in uppercase is", s.upper())

# Converts string to lowercase
s.lower()
print("the string in lowercase is", s.lower())

# Capitalizes the first character of the string
s.capitalize()
print("the string with first letter capitalized is", s.capitalize())

# Finds the first occurrence of a substring
s.find("World")
print("the first occurrence of the substring is at", s.find("World"))

# Replaces a substring with another string
s.replace("World", "Python")
print("the string after replacement is", s.replace("World", "Python"))

# Splits the string into a list
s.split(", ")
print("the string after splitting is", s.split(", "))

# Strips leading/trailing whitespace
s.strip()
print("the string after stripping is", s.strip())

# count the number of occurrences of a substring
s.count("l")
print("the number of occurrences of the substring is", s.count("l"))

# Checks if the string is alphanumeric
s.isalnum()
print("is the string alphanumeric?", s.isalnum())

```

the length of the string is 13
the string in uppercase is HELLO, WORLD!
the string in lowercase is hello, world!
the string with first letter capitalized is Hello, world!
the first occurrence of the substring is at 7
the string after replacement is Hello, Python!
the string after splitting is ['Hello', 'World!']
the string after stripping is Hello, World!
the number of occurrences of the substring is 3
is the string alphanumeric? False

3.3.3.2 Single quotes ' and double quotes " to define strings

in Python, you can use either single quotes (') or double quotes (") to define strings. Both are functionally equivalent, and you can choose based on preference or readability. Here's an example:

```
# Using single quotes
string1 = 'Hello, world!'
print(string1)

# Using double quotes
string2 = "Hello, world!"
print(string2)
```

Hello, world!
Hello, world!

When to use one over the other

- Single quotes (') are often preferred for simple strings without embedded quotes.
- Double quotes (") are useful when your string contains a single quote, as it avoids the need for escaping:

```
# Single quote in a double-quoted string
message = "It's a beautiful day!"
print(message)

# Double quote in a single-quoted string
message = 'He said, "Hello!"'
print(message)
```

```
It's a beautiful day!
He said, "Hello!"
```

Escaping quotes: If your string contains both single and double quotes, you can use the backslash (\) to escape them:

```
# Escaping single quotes in a single-quoted string
string_with_escape1 = 'It\'s a sunny day.'
print(string_with_escape1)

# Escaping double quotes in a double-quoted string
string_with_escape2 = "He said, \"Hello!\""
print(string_with_escape2)
```

```
It's a sunny day.
He said, "Hello!"
```

You can also use triple quotes (''' or """) for strings that span multiple lines or contain both types of quotes without escaping:

```
multi_line_string = """This string spans
multiple lines and can include 'single quotes' and "double quotes"."""

print(multi_line_string)
```

```
This string spans
multiple lines and can include 'single quotes' and "double quotes".
```

Define a *f*-string

```
language = "Python"
level = "beginner"
greeting1 = "I'm learning {} at a {} level.".format(language, level)
print(greeting1 )

greeting2 = f"I'm learning {language} at a {level} level."
print(greeting2)
```

```
I'm learning Python at a beginner level.
I'm learning Python at a beginner level.
```

String Concatenation: Using the + Operator * Use the + operator to join strings together.
* All operands must be strings; otherwise, you'll get a `TypeError`. * * Use `str()` to convert non-strings to strings when necessary.

```
# Basic Concatenation
greeting = "Hello"
name = "Alice"
message = greeting + ", " + name + "!"
print(message)

# Concatenating String Literals
print("Python" + " is " + "fun!")

# Concatenating with Variables
lang = "Python"
level = "beginner"
print("I'm learning " + lang + " as a " + level + " programmer.")
```

```
Hello, Alice!
Python is fun!
I'm learning Python as a beginner programmer.
```

String Repetition: String repetition is achieved using the * operator. It creates a new string by repeating an existing string a specified number of times.

```
# Creating Patterns
print("*" * 10)

*****

repeat_count = 4
print("Python! " * repeat_count)
```

```
Python! Python! Python! Python!
```

3.3.3.3 Integers

Integers are whole numbers, either positive or negative.

Commonly used Methods:

```

# Example:
n = -42

# Returns the absolute value
abs(n)
print("the absolute value of the number is", abs(n))

# Converts to binary string
bin(n)
print("the binary string of the number is", bin(n))

# Converts to hexadecimal string
hex(n)
print("the hexadecimal string of the number is", hex(n))

# Converts to octal string
oct(n)
print("the octal string of the number is", oct(n))

# Returns the power of a number
pow(n, 2) # n^2
print("the power of the number is", pow(n, 2))

# Checks if a number is an integer
isinstance(n, int)
print("is the number an integer?", isinstance(n, int))

```

```

the absolute value of the number is 42
the binary string of the number is -0b101010
the hexadecimal string of the number is -0x2a
the octal string of the number is -0o52
the power of the number is 1764
is the number an integer? True

```

3.3.3.4 Floats

Floats represent real numbers and are used for decimal or fractional values.

Commonly Used Methods:

```

# Example:
f = 3.14159

# Returns the absolute value
abs(f)
print("the absolute value of the number is", abs(f))

# Rounds to the nearest integer
round(f)
print("the number rounded to the nearest integer is", round(f))

# Converts to integer by truncating
int(f)
print("the number converted to integer is", int(f))

# Checks if a number is a float
isinstance(f, float)
print("is the number a float?", isinstance(f, float))

```

```

the absolute value of the number is 3.14159
the number rounded to the nearest integer is 3
the number converted to integer is 3
is the number a float? True

```

3.3.3.5 Booleans

Booleans represent logical values True or False.

Commonly Used Methods:

```

# Example:
b = True

# Converts to integer (True -> 1, False -> 0)
int(b)
print("the integer value of the boolean is", int(b))

# Converts to string
str(b)
print("the string value of the boolean is", str(b))

```

```
# Logical operations:
not b # Negates the boolean
print("the negation of the boolean is", not b)
```

```
the integer value of the boolean is 1
the string value of the boolean is True
the negation of the boolean is False
```

3.4 Variables

A **variable** is a container for storing data values. Variables in Python are **dynamically typed**, meaning you don't need to specify their type when declaring them.

3.4.1 Variable Declaration:

- You can create a variable by assigning a value to it using the = operator.

For example:

```
x = 10 # Integer
name = "Alice" # String
pi = 3.14 # Float
is_active = True # Boolean
```

3.4.2 Dynamic Typing:

- The type of a variable is determined by the value assigned to it.

For example:

```
x = 10 # x is an integer
y = x # y is also an integer
```

3.4.3 Variable Naming Rules:

- Names must start with a letter (a-z, A-Z) or an underscore (_).
- Names can only contain letters, numbers (0-9), and underscores.
- Names are case-sensitive (`name` and `Name` are different variables).
- Reserved keywords (e.g., `if`, `for`, `while`) cannot be used as variable names.

There are certain *reserved words* in python that have some meaning, and cannot be used as variable names. These reserved words are:

```
<IPython.core.display.Image object>
```

3.4.4 Best Practices:

[Python style guide](#): Please refer to the python style guide for best coding practices, such as naming variables, using spaces, tabs, and styling the different components of your code.

For example:

```
# use descriptive variable names:  
total_price = 100
```

```
# use snake_case for variable names  
user_age = 25
```

3.4.5 Checking Variable Types

You can use the `type()` function to check the type of a variable.

```
x = 10  
print(type(x))  
  
y = "Python"  
print(type(y))
```

```
<class 'int'>  
<class 'str'>
```


3.4.6 Practice exercise 4

Which of the following variable names are valid?

1. var.name
2. var9name
3. __varname
4. varname*

In the statements below, determine the variable type

1. value = "name"
2. constant = 7
3. another_const = "variable"
4. True_False = True

3.5 Assignment statements

Values are assigned to variables with the assignment statement (=). An assignment statement may have a constant or an expression on the right hand side of the (=) sign, and a variable name on the left hand side.

For example, the code lines below are assignment statements

```
var = 2
var = var + 3
```

3.6 Expressions

3.6.1 Mathematical Operations and Their Operators in Python

Python provides the following operators for performing mathematical operations:

1. **Exponentiation (**)**: Raises a number to the power of another.
 - Example: 2 ** 3 results in 8.
2. **Modulo (%)**: Returns the remainder of a division.
 - Example: 10 % 3 results in 1.

3. **Multiplication (*)**: Multiplies two numbers.
 - Example: $4 * 5$ results in 20.
4. **Division (/)**: Divides one number by another, resulting in a float.
 - Example: $10 / 2$ results in 5.0.
5. **Addition (+)**: Adds two numbers.
 - Example: $7 + 3$ results in 10.
6. **Subtraction (-)**: Subtracts one number from another.
 - Example: $9 - 4$ results in 5.

3.6.2 Operator Precedence in Python

The operators listed above are in **decreasing order of precedence**, meaning:

1. **Exponentiation (**) is evaluated first.**
2. **Modulo (%) is evaluated next.**
3. **Multiplication (*) follows.**
4. **Division (/)**, if present, has the same precedence as multiplication.
5. **Addition (+) and Subtraction (-)** are evaluated last, from left to right.

3.6.2.1 Example: Precedence in Action

Consider the expression: $2 + 3 \% 4 * 2$

To evaluate this, Python follows the precedence rules:

1. **Modulo (%) is evaluated first:**

```
3 % 4
```

3

Multiplication (*) is evaluated next:

```
3 * 2
```

6

Addition (+) is evaluated last:

```
2+6
```

8

Thus, the result of the expression `2 + 3 % 4 * 2` is 8.

3.6.2.2 Key Takeaways

- Precedence determines the order in which operations are performed in an expression.
- Parentheses () can be used to override the default precedence and control the order of evaluation.

```
result = (2 + 3) % (4 * 2)
print(result)
```

5

3.6.3 Practice exercise 5

Which of the following statements is an assignment statement:

1. `x = 5`
2. `print(x)`
3. `type(x)`
4. `x + 4`

What will be the result of the following expression:

```
1%2**3*2+1
```

3.7 User input

Python's in-built `input()` function is used to take input from the user during program execution. It reads a line of text entered by the user and returns it as a **string**.

```
# suppose we wish the user to onput their age:
age = input("Enter your age:")
```

The entered value is stored in the variable `age` and can be used for computation.

3.7.0.1 Key Point

- The `input()` is always returned as a string, even if the user enters a number.
- You can convert the input to other types (e.g., `int`, `float`) using type conversion functions.
- The program execution pauses until the user provides input.

3.7.1 Examples

```
# basic input
name = input("Enter your name: ")
print("Hello, " + name)
```

```
# using f-string for formatted output
name = input("Enter your name: ")
print(f"Hello, {name}!")
```

```
# To take numeric input, you need to convert the string to an appropriate data type:
age = int(input("Enter your age: "))
print(f"You will be {age + 1} years old next year.")
```

```
# input for calculating the area of a circle
radius = float(input("Enter the radius of the circle: "))
area = 3.14 * radius ** 2
print(f"The area of the circle is {area}")
```

3.7.2 Practice exercise 6

Ask the user to input their year of birth, and print their age.

3.8 Converting data types in Python

3.8.1 Why Convert Data Types in Python?

Data type conversion is essential in Python for several reasons:

- **Compatibility:** Some operations or functions require specific data types to work correctly.
 - Example: Performing arithmetic operations like addition or multiplication requires numeric types such as int or float. If the input is in another type, such as a string, it must be converted first.

```
# Example: Converting strings to numbers
price = "19.99"
tax = 0.07
total_price = float(price) * (1 + tax) # Convert string to float
print(total_price) # Output: 21.3893
```

21.3893

- **Data Processing:** When working with input data (e.g., user input, etc), the data may need to be converted to the appropriate type for further analysis.

Example: Converting strings to numbers to perform calculations

```
# Example: Arithmetic requires numeric types
num_str = "42"
result = int(num_str) + 10 # Converts the string "42" to integer
print(result) # Output: 52
```

52

- **Error Prevention:** Converting data types ensures consistency and prevents runtime errors caused by type mismatches.

```
# Example: Avoiding type mismatch errors
age = 25
message = "Your age is " + str(age) # Convert integer to string for concatenation
print(message) # Output: "Your age is 25"
```

Your age is 25

3.8.2 How to Convert Data Types in Python

Python provides several built-in functions for type conversion.

Common Conversion Functions:

Function	Description	Example
<code>int()</code>	Converts to an integer (from float or string)	<code>int("42") → 42</code>
<code>float()</code>	Converts to a float	<code>float("3.14") → 3.14</code>
<code>str()</code>	Converts to a string	<code>str(42) → "42"</code>
<code>bool()</code>	Converts to a boolean	<code>bool(1) → True</code>

However, in some cases, mathematical operators such as `+` and `*` can be applied on strings. The operator `+` concatenates multiple strings, while the operator `*` can be used to concatenate a string to itself multiple times:

```
"Hi" + " there!"
```

```
'Hi there!'
```

```
"5" + '3'
```

```
'53'
```

```
"5"*8
```

```
'55555555'
```

3.9 Errors and Exceptions

Errors and Exceptions are common while writing and executing Python code.

3.9.1 Syntax errors

Syntax errors occur if the code is written in a way that it does not comply with the rules / standards / laws of the language (python in this case). It occurs when the Python parser encounters invalid syntax.

For example, suppose a value is assigned to a variable as follows:

```
9value = 2
```

The above code when executed will indicate a syntax error as it violates the rule that a variable name must not start with a number.

```
# another example
print("Hello World")
```

Solution: Fix the syntax issue by ensuring correct punctuation or structure.

3.9.2 Exceptions

Even if a statement or expression is syntactically correct, it may cause an error when an attempt is made to execute it. Errors detected during execution are called exceptions and are not unconditionally fatal:

Exceptions come in different types, and the type is printed as part of the message: below are the common ones:

- Misspelled or incorrectly capitalized variable and function names
- Attempts to perform operations (such as math operations) on data of the wrong type (ex. attempting to subtract two variables that hold string values)
- Dividing by zero
- Attempts to use a type conversion function such as `int` on a value that can't be converted to an `int`

For example, suppose a number is multiplied as follows:

```
multiplication_result = misy * 4
```

```
NameError: name 'misy' is not defined
```

```
-----
NameError                                Traceback (most recent call last)
Cell In[29], line 1
----> 1 multiplication_result = misy * 4
NameError: name 'misy' is not defined
```

The above code is syntactically correct. However, it will generate an error as the variable `misy` has not been defined as a number.

```
int("abc")
```

```
ValueError: invalid literal for int() with base 10: 'abc'
```

```
-----  
ValueError                                Traceback (most recent call last)  
Cell In[25], line 1  
----> 1 int("abc")  
ValueError: invalid literal for int() with base 10: 'abc'
```

```
print("2" + 3)
```

```
TypeError: can only concatenate str (not "int") to str
```

```
-----  
TypeError                                Traceback (most recent call last)  
Cell In[24], line 1  
----> 1 print("2" + 3)  
TypeError: can only concatenate str (not "int") to str
```

```
print(10 / 0)
```

```
ZeroDivisionError: division by zero
```

```
-----  
ZeroDivisionError                        Traceback (most recent call last)  
Cell In[27], line 1  
----> 1 print(10 / 0)  
ZeroDivisionError: division by zero
```

3.9.3 Exception Handling

When an error occurs, or exception as we call it, Python will normally stop and generate an error message.

If we suspect that some lines of code may produce an error, we can put them in a **try** block, and if an error does occur, we can use the **except** block to instead execute an alternative piece of code. This way the program will not stop if an error occurs within the **try** block, and instead will be directed to execute the code within the **except** block.

These exceptions can be handled Using the Try-Except Blocks

```
try:  
    print(10 / 0)  
except:  
    print("Cannot divide by zero!")
```


Cannot divide by zero!

Since the try block raises an error, the except block will be executed. Without the try block, the program will crash and raise an error:

The finally block, if specified, will be executed regardless if the try block raises an error or not.

```
try:
    print(10 / 0)
except:
    print("Cannot divide by zero!")
finally:
    print("This will always execute.")
```

Cannot divide by zero!
This will always execute.

3.9.4 Practice exercise 7

Suppose we wish to compute tax using the income and the tax rate. Identify the type of error from amongst syntax error, semantic error and run-time error in the following pieces of code.

```
income = 2000
tax = .08 * Income
print("tax on", income, "is:", tax)
```

```
income = 2000
tax = .08 x income
print("tax on", income, "is:", tax)
```

```
income = 2000
tax = .08 ** income
print("tax on", income, "is:", tax)
```

3.9.5 Practice exercise 8

Input an integer from the user. If the user inputs a valid integer, print whether it is a multiple of 3. However, if the user does not input a valid integer, print a message saying that the input is invalid.

```

num = input("Enter an integer:")

#The code lines within the 'try' block will execute as long as they run without error
try:
    #Converting the input to integer, as user input is a string
    num_int = int(num)

    #checking if the integer is a multiple of 3
    if num_int % 3 == 0:
        print("Number is a multiple of 3")
    else:
        print("Number is not a multiple of 3")

#The code lines within the 'except' block will execute only if the code lines within the 'try'
except:
    print("Input must be an integer")

```

Input must be an integer

3.9.6 Semantic errors (bugs)

Semantic errors occur when the code executes without an error being indicated by the compiler. However, it does not work as intended by the user. For example, consider the following code of multiplying the number 6 by 3: `x = '6' * 3`. If it was intended to multiply the number 6, then the variable `x` should have been defined as `x=6` so that `x` has a value of type `integer`. However, in the above code `6` is a `string` type value. When a `string` is multiplied by an integer, say n , it concatenates with itself n times.

3.9.7 Practice exercise 9

The formula for computing final amount if one is earning compound interest is given by:

$$A = P \left(1 + \frac{r}{n} \right)^{nt},$$

where:

P = Principal amount (initial investment),

r = annual nominal interest rate,

n = number of times the interest is computed per year,

t = number of years

Write a Python program that assigns the principal amount of \$10000 to variable P , assign to n the value 12, and assign to r the interest rate of 8%. Then have the program prompt the user for the number of years t that the money will be compounded for. Calculate and print the final amount after t years.

What is the amount if the user enters t as 4 years?

4 Control flow tools

<IPython.core.display.Image object>

A [control flow statement](#) in a computer program determines the individual lines of code to be executed and/or the order in which they will be executed. In this chapter, we'll learn about 3 types of control flow statements:

1. if-elif-else
2. for and while loops
3. break, continue, and pass statements

4.1 Indentation in Python

Syntax: Python uses indentation to identify the code to be executed if a condition is true. All the code indented within a condition is executed if the condition is true.

4.1.1 What is Indentation?

- Indentation refers to the spaces or tabs at the beginning of a line of code.
- In Python, indentation is **mandatory** and is used to define the structure of the code, such as blocks in loops, conditionals, functions, and classes.
- Unlike some other programming languages, Python does not use braces {} or keywords like **begin** and **end** to define blocks of code.

4.1.2 Rules for Indentation

1. **Consistency is Key:**

- Use either spaces or tabs for indentation, but **do not mix them** in the same file.
- The recommended standard is to use **4 spaces** per indentation level (PEP 8).

2. **Indentation Levels:**

- Each block of code under a statement (e.g., **if**, **for**, **while**, **def**) must be indented one level deeper than the statement itself.

<IPython.core.display.Image object>

4.1.3 Examples

```
# Example of proper indentation
def greet(name):
    if name:
        print(f"Hello, {name}!")
    else:
        print("Hello, World!")
```

4.2 Conditional execution

The first type of control flow statement is **if-elif-else**. This statement helps with conditional execution of code, i.e., the piece of code to be executed is selected based on certain condition statements(s).

<IPython.core.display.Image object>

4.2.1 Comparison operators

For testing if conditions are true or false, first we need to learn the operators that can be used for comparison. For example, suppose we want to check if two objects are equal, we use the **==** operator:

```
5 == 6
```

False

Make sure you can differentiate between the **==** and **=** operators: - **==**: This is a **comparison operator**, used to compare two values and return a Boolean result (**True** or **False**). - **=**: This is an **assignment operator**, used to assign values to variables.

```
x = "hi"
y = "hi"
x == y
```

True

Below are the python comparison operators and their meanings.

Python code	Meaning
<code>x == y</code>	Produce True if ... x is equal to y
<code>x != y</code>	... x is not equal to y
<code>x > y</code>	... x is greater than y
<code>x < y</code>	... x is less than y
<code>x >= y</code>	... x is greater than or equal to y
<code>x <= y</code>	... x is less than or equal to y

4.2.2 Logical Operators in Conditional Statements

Logical operators are used to **combine multiple conditions** in a conditional statement, allowing for more complex decision-making. Python provides three logical operators:

4.2.2.1 and

- **Description:** Returns True if both conditions are True.
- **Example:**

```
x = 5
if x > 0 and x < 10:
    print("x is a positive single-digit number.")
```

x is a positive single-digit number.

4.2.2.2 or

- **Description:** Returns True if at least one condition is True.
- **Example:**

```
x = -5
if x < 0 or x > 10:
    print("x is either negative or greater than 10.")
```

x is either negative or greater than 10.

4.2.2.3 not

- **Description:** Returns the negation of a condition (**True** becomes **False** and vice versa).
- **Example:**

```
x = 5
if not (x < 0):
    print("x is not negative.")
```

x is not negative.

4.2.3 if-elif-else statement

The **if-elif-else** statements can check several conditions, and execute the code corresponding to the condition that is true. Note that there can be as many **elif** statements as required.

Syntax: Python uses indentation to identify the code to be executed if a condition is true. All the code indented within a condition is executed if the condition is true.

Example: Input an integer. Print whether it is positive or negative.

```
number = input("Enter a number:") #Input an integer
number_integer = int(number)      #Convert the integer to 'int' datatype
if number_integer > 0:             #Check if the integer is positive
    print("Number is positive")
else:
    print("Number is negative")
```

Number is positive

In the above code, note that anything entered by the user is taken as a string datatype by python. However, a string cannot be positive or negative. So, we converted the number input by the user to integer to check if it was positive or negative.

There may be multiple statements to be executed if a condition is true. See the example below.

Example: Input a number. Print whether it is positive, negative or zero. If it is negative, print its absolute value.

```

number = input("Enter a number:")
number_integer = int(number)
if number_integer > 0:
    print("Number is positive")
elif number_integer == 0:
    print("Number is zero")
else:
    print("Number is negative")
    print("Absolute value of number = ", abs(number_integer))

```

Number is positive

4.2.4 Practice exercise 1

Input a number. Print whether its odd or even.

Solution:

```

num = int(input("Enter a number: "))
if num%2 == 0:          #Checking if the number is divisible by 2
    print("Number is even")
else:
    print("Number is odd")

```

Number is odd

4.2.5 Practice exercise 2

4.2.5.1

Ask the user to enter their exam score. Print the grade based on their score as follows:

Score	Grade
(90,100]	A
(80,90]	B
[0,80]	C

If the user inputs a score which is not a number in [0,100], print invalid entry.

Solution:


```

score = input("Enter exam score:")
try:

    #As exam score can be a floating point number (such as 90.65), we need to use 'float' in
    score_num = float(score)
    if score_num > 90 and score_num <= 100:
        print("Grade: A")
    elif score_num > 80 and score_num <= 90:
        print("Grade: B")
    elif score_num >= 0 and score_num <= 80:
        print("Grade: C")
    else:
        print("Invalid score")          #If a number is less than 0 or more than 100
except:
    print("Invalid input")              #If the input is not a number

```

Grade: C

4.2.5.2

Nested if-elif-else statements: This question will lead you to create nested if statements, i.e., an if statement within another if statement.

Think of a number in [1,5]. Ask the user to guess the number.

- If the user guesses the number correctly, print “Correct in the first attempt!”, and stop the program. Otherwise, print “Incorrect! Try again” and give them another chance to guess the number.
- If the user guesses the number correctly in the second attempt, print “Correct in the second attempt”, otherwise print “Incorrect in both the attempts, the correct number is:”, and print the correct number.

Solution:

```

#Let us say we think of the number. Now the user has to guess the number in two attempts.
rand_no = 3
guess = input("Guess the number:")
if int(guess)==rand_no:
    print("Correct in the first attempt!")

#If the guess is incorrect, the program will execute the code block below
else:

```

```
guess = input("Incorrect! Try again:")
if int(guess) == rand_no:
    print("Correct in the second attempt")
else:
    print("Incorrect in the both the attempts, the correct number was:", rand_no)
```

Incorrect in the both the attempts, the correct number was: 3

4.3 Loops in Python

Python provides two types of loops: **for** and **while** loops. Loops are used to execute a block of code repeatedly until a certain condition is met.

4.3.1 Using range() in for Loops

The `range()` function is commonly used with loops in Python to generate a sequence of numbers. It is particularly useful with for loops.

4.3.1.1 Syntax of range()

`range(start, stop, step)`

- **start:** (Optional) The starting value of the sequence (default is 0).
- **stop:** (Required) The endpoint of the sequence (exclusive).
- **step:** (Optional) The difference between each number in the sequence (default is 1).

Using the `range()` function, the **for** loop can iterate over a sequence of numbers. See the examples below.

```
for i in range(5):
    print(i)
```

0
1
2
3
4

Note that the last element is one less than the integer specified in the `range()` function.

```
# specify start and stop
for i in range(5, 10):
    print(i)
```

5
6
7
8
9

```
# use step
for i in range(0, 10, 2):
    print(i)
```

0
2
4
6
8

```
# A negative step will count down
for i in range(10, 0, -2):
    print(i)
```

10
8
6
4
2

```
# if start is greater than stop, the range will generate an empty sequence
for i in range(10, 0):
    print(i)
```

Example: Print the first n elements of the [Fibonacci sequence](#), where n is an integer input by the user, such that $n > 2$. In a fibonacci sequence, each number is the sum of the preceding two numbers, and the sequence starts from 0,1. The sequence is as follows:

0,1,1,2,3,5,8,13,....

```

n = int(input("Enter number of elements:"))

#Initializing the sequence to start from 0, 1
n1, n2 = 0, 1

#Printing the first two numbers of the sequence
print(n1)
print(n2)

for i in range(n-2): #Since two numbers of the sequence are already printed, n-2 numbers are

    #Computing the next number of the sequence as the summation of the previous two numbers
    n3 = n1 + n2
    print(n3)

    #As 'n3' is already printed, it is no longer the next number of the sequence.
    #Thus, we move the values of the variables n1 and n2 one place to the right to compute the next number
    n1 = n2
    n2 = n3

print("These are the first", n, "elements of the fibonacci series")

```

```

0
1
1
2
3
5
8
13
21
34
55
89
144
233
377
610
987
1597
2584
4181

```

6765
10946
17711
28657
46368
75025
121393
196418
317811
514229
832040
1346269
2178309
3524578
5702887
9227465
14930352
24157817
39088169
63245986
102334155
165580141
267914296
433494437
701408733
1134903170
1836311903
2971215073
4807526976
7778742049
12586269025
20365011074
32951280099
53316291173
86267571272
139583862445
225851433717
365435296162
591286729879
956722026041
1548008755920
2504730781961
4052739537881

```
6557470319842
10610209857723
17167680177565
27777890035288
44945570212853
72723460248141
117669030460994
190392490709135
308061521170129
498454011879264
806515533049393
1304969544928657
2111485077978050
3416454622906707
5527939700884757
```

These are the first 78 elements of the fibonacci series

As in the `if-elif-else` statement, the `for` loop uses indentation to indicate the piece of code to be run repeatedly.

```
# nested with range
for i in range(5):
    for j in range(5):
        print(i, j)
```

```
0 0
0 1
0 2
0 3
0 4
1 0
1 1
1 2
1 3
1 4
2 0
2 1
2 2
2 3
2 4
3 0
3 1
```

```
3 2
3 3
3 4
4 0
4 1
4 2
4 3
4 4
```

4.3.2 while loop

With a **while** loops, a piece of code is executed repeatedly until **certain condition(s)** hold.

Example: Print all the elements of the [Fibonacci sequence](#) less than n , where n is an integer input by the user, such that $n > 2$. In a fibonacci sequence, each number is the sum of the preceding two numbers, and the sequence starts from 0,1. The sequence is as follows:

0,1,1,2,3,5,8,13,....

```
# Get user input for the Fibonacci sequence limit
n = int(input("Enter the value of n: "))

# Initializing the sequence to start from 0 and 1
n1, n2 = 0, 1

# Print the first number of the sequence
print(n1)

# Generate Fibonacci numbers less than n
while n2 < n:
    print(n2) # Print the next number in the sequence

    # Compute the next number in the sequence as the sum of the previous two numbers
    n3 = n1 + n2

    # Update values: shift n1 to n2, and n2 to the newly computed value n3
    n1, n2 = n2, n3

# Print completion message
print(f"These are all the elements of the Fibonacci series less than {n}.")
```

0

1
1
2
3
5
8
13
21

These are all the elements of the fibonacci series less than 23

Let's take it a step further: if the user enters an invalid input, the Python program will repeatedly prompt them until they provide a valid one, using a while loop

```
# Prompt the user for a valid positive integer
while True:
    user_input = input("Enter a positive integer (n): ")
    if user_input.isdigit(): # Check if input is numeric
        n = int(user_input)
        if n > 0: # Ensure the number is positive
            break # Exit the loop if input is valid
        else:
            print("Error: Please enter a number greater than 0.")
    else:
        print("Error: Invalid input. Please enter a positive integer.")

# Initializing the sequence to start from 0 and 1
n1, n2 = 0, 1

# Print the first number of the sequence
print(n1)

# Generate Fibonacci numbers less than n
while n2 < n:
    print(n2) # Print the next number in the sequence

    # Compute the next number in the sequence as the sum of the previous two numbers
    n3 = n1 + n2

    # Update values: shift n1 to n2, and n2 to the newly computed value n3
    n1, n2 = n2, n3

# Print completion message
print(f"These are all the elements of the Fibonacci series less than {n}.")
```

Ensures valid input using a while loop:

- Keeps prompting the user until they enter a **positive integer**.
- Uses `.isdigit()` to check if input is numeric.
- Converts input to `int` and ensures it's greater than 0.

4.3.3 Practice exercise 3

4.3.3.1

Write a program that identifies whether a number input by the user is prime or not.

Solution:

```
number = int(input("Enter a positive integer:"))

#Defining a variable that will have a value of 0 if there are no divisors
num_divisors = 0

#Checking if the number has any divisors from 2 to half of the number
for divisor in range(2,int(number/2+1)):
    if number % divisor == 0:

        #If the number has a divisor, setting num_divisors to 1, to indicate that the number is not prime
        num_divisors = 1

        #If a divisor has been found, there is no need to check if the number has more divisors
        #Even if the number has a single divisor, it is not prime. Thus, we 'break' out of the loop
        #If you don't 'break', your code will still be correct, it will just do some unnecessary calculations
        break

#If there are no divisors of the number, it is prime, else not prime
if num_divisors == 0:
    print("Prime")
else:
    print("Not prime")
```

Not prime

4.3.3.2

Update the program above to print the prime numbers starting from 2, and less than n where n is a positive integer input by the user.

Solution:

```
n = int(input("Enter a positive integer:"))

#Defining a variable - number_iterator. We will use this variable to iterate over all integers
#While iterating over each integer from 2 to n, we will check if the integer is prime or not
number_iterator = 2

print(number_iterator) #Since '2' is a prime number, we can print it directly (without checking)

#Continue to check for prime numbers until n (but not including n)
while(number_iterator < n):

    #After each check, increment the number_iterator to check if the next integer is prime
    number_iterator = number_iterator + 1

    #Defining a variable that will have a value of 0 if there are no divisors
    num_divisors = 0

    #Checking if the integer has any divisors from 2 to half of the integer being checked
    for divisor in range(2,int(number_iterator/2 + 1)):
        if number_iterator % divisor == 0:

            #If the integer has a divisor, setting num_divisors to 1, to indicate that the number is not prime
            num_divisors = 1

            #If a divisor has been found, there is no need to check if the integer has more divisors
            #Even if the integer has a single divisor, it is not prime.
            #Thus, we 'break' out of the loop that checks for divisors
            break

    #If there are no divisors of the integer being checked, the integer is a prime number, and we print it
    if num_divisors == 0:
        print(number_iterator)
```

2
3

5
7
11
13
17
19
23
29
31
37
41
43

4.4 Control flow statements

They are used to influence the flow of execution in loops or blocks of code. Python provides three such statements: `break`, `continue`, and `pass`.

4.4.1 `break` statement

The `break` statement is used to **exit a loop prematurely** before it has iterated through all elements or completed its condition. It is commonly used in both `for` and `while` loops to stop the execution of the loop when a specific condition is met.

For example

```
for i in range(10):  
    if i == 5:  
        print("Breaking the loop at i =", i)  
        break  
    print(i)
```

0
1
2
3
4
Breaking the loop at i = 5

4.4.2 Practice exercise 4

Write a program that finds and prints the largest factor of a number input by the user. Check the output if the user inputs 133.

Solution:

```
num = int(input("Enter an integer:"))

#Looping from the half the integer to 0 as the highest factor is likely to be closer to half
for i in range(int(num/2) + 1, 0, -1):
    if num%i == 0:
        print("Largest factor = ", i)

        #Exiting the loop if the largest integer is found
        break
```

Largest factor = 1

4.4.3 continue statement

The continue statement is used to **skip the current iteration of a loop** and move to the next iteration. Unlike the break statement, it does not terminate the loop but allows the loop to continue running.

For example, consider the following code:

```
for i in range(5):
    if i == 3:
        print(f"Skipping iteration {i}")
        continue
    print(f"Processing {i}")
```

Processing 0
Processing 1
Processing 2
Skipping iteration 3
Processing 4

The continue statement skips the iteration when `i == 3` and moves to the next iteration.

4.4.4 Practice exercise 5:

Write a program that asks the user the question, “How many stars are in the Milky Way (in billions)?”. If the user answers 100, the program should print correct, and stop. However, if the user answers incorrectly, the program should print “incorrect”, and ask them if they want to try again. The program should continue to run until the user answers correctly, or they want to stop trying.

```
#Defining an infinite while loop as the loop may need to run indefinitely if the user keeps a
while True:
    answer = input("How many stars are there in the Milky Way? ")
    if answer == '100':
        print("Correct")

        #Exiting the loop if the user answers correctly
        break
    else:
        print("Incorrect")
        try_again = input("Do you want to try again? (Y/N) ")
        if try_again == 'Y':

            #Continuing with the infinite loop if the user wants to try again
            continue
        else:

            #Exiting the infinite loop if the user wants to stop tryinh
            break
```

Incorrect

4.4.5 pass statement

The **pass** statement in Python is a **null statement**. It serves as a placeholder and does nothing when executed. It is often used in situations where a statement is syntactically required, but no action is intended, or the code is yet to be implemented. In Chapter 4, we will use this statement when we explore user-defined functions.

4.5 Loops with strings

Strings in Python are sequences of characters. You can use loops to iterate over strings and perform various operations on each character or a subset of the string.

Consider the following string:

```
sentence = "She sells sea shells on the sea shore"
```

The i^{th} character of the string can be retrieved by its index. For example, the first character of the string `sentence` is:

```
sentence[0]
```

```
'S'
```

A `for` loop iterates over each character in a string, one at a time.

```
for char in sentence:  
    print(char, end=" ")
```

```
S h e   s e l l s   s e a   s h e l l s   o n   t h e   s e a   s h o r e
```

A `while` loop can be used to iterate over a string by index.

```
index = 0  
while index < len(sentence):  
    print(sentence[index], end=" ")  
    index += 1
```

```
S h e   s e l l s   s e a   s h e l l s   o n   t h e   s e a   s h o r e
```

Slicing a string:

A part of the string can be sliced by passing the starting index (say `start`) and the stopping index (say `stop`) as `start:stop` to the index operator `[]`. This is called slicing a string. For a string `S`, the characters starting from the index `start` upto the index `stop`, but not including `stop`, can be sliced as `S[start:stop]`.

For example, the slice of the string `sentence` from index 4 to index 9, but not including 9 is:

```
sentence[4:9]
```

```
'sells'
```

Example: Counting characters

Input a string, and count and print the number of “t”s.

```
str1 = input("Enter a sentence:")

#Initializing a variable 'count_t' which will store the number of 't's in the string
count_t = 0

#Iterating over the entire length of the string.
#The length of the string is given by the len() function
for i in range(len(str1)):

    #If the ith character of the string is 't', then we count it
    if str1[i] == 't':
        count_t = count_t + 1

print("Number of 't's in the str1 = ", count_t)
```

Number of 't's in the str1 = 2

Another way to achieve this is to use count()

```
str1.count('t')
```

2

4.5.1 Practice exercise 6

Checking for a Substring

Write a program that asks the user to input a string, and print the number of “the”s in the string.

```
str2 = input("Enter a sentence:")

#Defining a variable to store the count of the word 'the'
count_the = 0

#Looping through the entire length of the string except the last 3 letters.
#As we are checking three letters at a time starting from the index 'i', the last 3 letters o
```

```
for i in range(len(str2) - 3):  
  
    #Slicing 3 letters of the string and checking if they are 'the'  
    if str2[i:(i+3)] == 'the':  
  
        #Counting the words that are 'the'  
        count_the = count_the + 1  
print("Number of 'the's in the str2 = ", count_the)
```

Number of 'the's in the str2 = 3

```
# using the count method  
str2.count('the')
```

3

5 Functions

<IPython.core.display.Image object>

5.1 Function Definition

Functions are the fundamental building blocks of any Python program. They are organized blocks of reusable code designed to perform a specific task. A function can take one or more inputs (parameters), execute a block of code, and optionally return one or more values.

5.1.1 Why Use Functions?

Functions allow developers to write modular, reusable, and efficient code. Instead of duplicating the same logic multiple times, functions let you define the logic once and call it wherever needed.

5.2 Advantages of Functions

1. Increases Modularity

- Functions allow the program to be divided into smaller, manageable parts, making it easier to understand, implement, and maintain.

2. Reduces Redundancy

- By defining a function once, you avoid rewriting the same code multiple times. Simply call the function when needed.

3. Maximizes Code Reusability

- Functions can be used as many times as necessary, enabling you to reuse your code efficiently and reducing overall development effort.

4. Improves Code Readability

- Dividing a large program into smaller functions improves the clarity and readability of the code, making it easier to debug and maintain.

5.3 Types of Functions

There are two types of functions in python:

- **Predefined Functions** - These are built-in functions in python.
- **User-Defined Functions** - these types of functions are defined by the user to perform any specific task

5.3.1 Functions

These are built-in functions that perform common tasks. Built-in functions come from two main sources:

- Python Standard Libraries
- Third-Party Libraries

5.3.1.1 Python Standard Library

The Python Standard Library is an umbrella term for all the modules (A module is a file containing Python code (functions, classes, variables) that can be reused in your programs) and packages that come with Python, including both built-in modules (e.g., `__builtins__`) and other modules that require importing. Think of the standard library as a toolbox, with some tools always on the table (built-in) and others stored in drawers (import-required). Built-in functions like `print()`, `len()`, and `type()` are available directly without needing to import anything. They are part of Python's built-in namespace, which is loaded into memory when Python starts.

Many modules in the Python Standard Library, like `math`, `os`, or `datetime`, are not automatically loaded to keep the startup time and memory usage low. To access functions or classes from these modules, you need to explicitly import them using the `import` keyword.

Let's see different ways to import modules next

- Basic Import

```
import math
# To use a function from the module, preface it with random followed by a dot, and then the :
print(math.sqrt(16))
```

4.0

- Import Specific Functions or Classes

```
# import only sqrt function from math module
from math import sqrt, pi
print(sqrt(25))
```

5.0

- Import with Alias:

```
import numpy as np
print(np.array([1, 2, 3]))
```

[1 2 3]

- Wildcard Import (Not Recommended):

```
from math import *
print(sin(1))
```

0.8414709848078965

This way imports every function from the module. You should usually **avoid** doing this, as the module may contain some names that will interfere with your own variable names. For instance if your program uses a variable called `total` and you import a module that contains a function called `total`, there can be problems. In contrast, the first way imports an entire module in a way that will not interfere with your variable names. To use a function from the module, preface it with the module name followed by a dot

Location: Usually, import statements go at the beginning of the program, but there is no restriction. They can go anywhere as long as they come before the code that uses the module.

5.3.1.2 Useful Modules

Here's a list of commonly used and useful modules from the Python Standard Library:

- **os:** For interacting with the operating system, such as file paths and environment variables.
- **sys:** For interacting with the Python runtime environment
- **re:** For regular expressions and pattern matching
- **math:** For mathematical functions and constants
- **random:** For generating random numbers.
- **datetime:** For working with dates and times
- **time:** For measuring time or introducing delays.

5.3.1.2.1 Random Numbers

Generating random numbers is very useful in python for performing simulations (we'll see in later chapters). The library `random` is used to generate random numbers such as integers, real numbers based on different probability distributions, etc.

Below is an example of using the `randint()` function of the library for generating random numbers in $[a, b]$, where `a` and `b` are integers.

```
import random as rm
rm.randint(5,10) #This will generate a random number in [5,10]
```

7

The random number will be different every time we run the program.

5.3.1.2.2 Math Functions

Python's math module is part of the standard library and provides access to common mathematical functions and constants. You can use these functions for tasks such as computing square roots, trigonometric operations, logarithms, factorials, and more.

For example:

```
from math import sin, pi
print(pi)
print(pi/2)
print('sin(pi/2) =', sin(pi/2))
```

```
3.141592653589793
1.5707963267948966
sin(pi/2) = 1.0
```

5.3.1.3 Getting Help from Python on a Module

There is documentation built into Python. To get help on the `random` module

```
dir(rm)
```

```
['BPF',  
 'LOG4',  
 'NV_MAGICCONST',  
 'RECIP_BPF',  
 'Random',  
 'SG_MAGICCONST',  
 'SystemRandom',  
 'TWOPI',  
 '_ONE',  
 '_Sequence',  
 '__all__',  
 '__builtins__',  
 '__cached__',  
 '__doc__',  
 '__file__',  
 '__loader__',  
 '__name__',  
 '__package__',  
 '__spec__',  
 '_accumulate',  
 '_acos',  
 '_bisect',  
 '_ceil',  
 '_cos',  
 '_e',  
 '_exp',  
 '_fabs',  
 '_floor',  
 '_index',  
 '_inst',  
 '_isfinite',  
 '_lgamma',  
 '_log',  
 '_log2',  
 '_os',  
 '_pi',  
 '_random',  
 '_repeat',  
 '_sha512',  
 '_sin',  
 '_sqrt',  
 '_test',  
 '_test_generator',
```

```

'_urandom',
'_warn',
'betavariate',
'binomialvariate',
'choice',
'choices',
'expovariate',
'gammavariate',
'gauss',
'getrandbits',
'getstate',
'lognormvariate',
'normalvariate',
'paretovariate',
'randbytes',
'randint',
'random',
'randrange',
'sample',
'seed',
'setstate',
'shuffle',
'triangular',
'uniform',
'vonmisesvariate',
'weibullvariate']

```

This provides a list of all the functions and variables in the `random` module. You can ignore entries that start with underscores, as they are typically used internally. To get help on a specific function, such as the `uniform` function, you can type:

```
help(rm.uniform)
```

Help on method uniform in module random:

`uniform(a, b)` method of `random.Random` instance

Get a random number in the range `[a, b)` or `[a, b]` depending on rounding.

The mean (expected value) and variance of the random variable are:

$$E[X] = (a + b) / 2$$

$$\text{Var}[X] = (b - a) ** 2 / 12$$

For a comprehensive overview of the entire `math` module, type:

```
# help(rm) #This will give you all the functions available in the random module
```

I encourage you to explore the documentation for a deeper understanding, especially when you need to use a module but are unsure how to get started.

5.3.2 Practice exercise 1

- Can you use `math.sqrt(16)` without importing the `math` module? Why or why not?
- Identify whether the following functions require importing a module:
 - `abs()`
 - `random.randint()`
 - `time.sleep()`

5.3.3 Practice exercise 2

Generate a random integer between `[-5,5]`. Do this 10,000 times. Find the **mean** of all the 10,000 random numbers generated.

5.3.3.1 Third-Party Python libraries

Other than the Python Standard Library, Python has hundreds of thousands of additional libraries that provide a wealth of useful functions. Since Python is an open-source platform, these libraries are contributed by developers from around the world. Some of the most popular libraries in data science and their purposes are listed below:

1. NumPy: Performing numerical operations and efficiently storing numerical data.
2. Pandas: Reading, cleaning and manipulating data.
3. Matplotlib, Seaborn: Visualizing data.
4. SciPy: Performing scientific computing such as solving differential equations, optimization, statistical tests, etc.
5. Scikit-learn: Data pre-processing and machine learning, with a focus on prediction.
6. Statsmodels: Developing statistical models with a focus on inference

Before you can use them, you need to **install** each library and then **import** it in your code.

A library can be imported using the **import** keyword after it has been successfully installed. For example, the NumPy library can be imported as:

```
import numpy as np
```

Using the `as` keyword, the NumPy library has been given the name `np`. All the functions and attributes of the library can be called using the `'np.'` prefix. For example, let us generate a sequence of whole numbers upto 10 using the NumPy function `arange()`:

```
np.arange(8)
```

```
array([0, 1, 2, 3, 4, 5, 6, 7])
```

You will use these libraries in the upcoming data science courses.

5.4 User-defined Functions

A user-defined function is a function created by the user in Python to perform a specific task. Unlike built-in functions (like `print()` or `len()`), user-defined functions allow you to define **custom functionality tailored to your program's needs**.

The image below provides a helpful breakdown of a Python function definition with labels for each component.

```
<IPython.core.display.Image object>
```

5.4.1 Key Components (Based on the Diagram)

1. **def keyword**: Indicates the start of a function definition.
2. **Function name**: A descriptive name for the function, following Python naming conventions.
3. **Parameters**: Variables passed into the function inside parentheses (`x, y` in the example). These are optional.
4. **Colon (:)**: Signals the end of the function header and the start of the body.
5. **Docstring**: A multi-line string (optional) that describes the purpose and functionality of the function.
6. **Function body**: Contains the logic and statements of the function.
7. **return statement**: Outputs a result back to the caller. This is optional.

Example


```
# define a function
def my_function():
    print("Hello from a function")
```

5.4.2 Functions are lazy

Functions are designed to be reusable. They don't run until explicitly called, so their behavior can be invoked multiple times,

Call the `my_function` to execute it

```
# Simply use the function's name followed by parentheses
my_function()

# call the function again
my_function()
```

```
Hello from a function
Hello from a function
```

The function was called twice, it printed out the information twice.

5.4.3 Arguments and Parameters in a Function

The terms arguments and parameters are often used interchangeably but have distinct meanings in the context of functions in Python.

5.4.3.1 Parameters:

- **Definition:** Parameters are the variables listed in a function's definition. They act as placeholders that specify the input a function can accept.
- **When Used:** Defined when you write the function.
- **Example:**

```
def greet_user(name): # 'name' is the parameter
    print(f"Hello, {name}!")
```

5.4.3.2 Arguments:

- **Definition:** Arguments are the actual values or data you pass to a function when you call it. These values are assigned to the function's parameters.
- **When Used:** Provided when you invoke (call) the function.
- **Example:**

```
greet_user("Alice") # "Alice" is the argument
```

Hello, Alice!

Another Example:

```
# Function definition with parameters
def add_numbers(a, b):
    return a + b

# Function call with arguments
result = add_numbers(5, 3)

print(result)
```

8

Understanding the distinction between **parameters** and **arguments** is crucial for writing clear and effective functions in Python.

5.4.4 Type of Arguments in Python

5.4.4.1 Required Arguments

These are the arguments that must be provided when the function is called. If they are missing, Python will raise a **TypeError**.

```
def greet_user(name):
    print(f"Hello, {name}!")

# Call with a required argument
greet_user("Alice") # Output: Hello, Alice!
```

```
# Call without an argument will raise an error
# greet_user() # TypeError: greet_user() missing 1 required positional argument: 'name'
```

Hello, Alice!

5.4.4.2 Keyword Arguments

These allow you to specify arguments by their parameter name. This makes your code more readable and avoids confusion, especially when dealing with multiple arguments.

```
def describe_person(name, age):
    print(f"{name} is {age} years old.")

# Call with keyword arguments
describe_person(name="Bob", age=30) # Output: Bob is 30 years old.
describe_person(age=25, name="Alice") # Output: Alice is 25 years old.

# Call without keywords (positional)
describe_person("Charlie", 40) # Output: Charlie is 40 years old.
describe_person(35, "David") # Output: 35 is David years old.
```

Bob is 30 years old.
Alice is 25 years old.
Charlie is 40 years old.
35 is David years old.

5.4.4.3 Default Arguments

These are parameters that have default values. If no argument is provided during the function call, the default value is used.

```
def greet(name="Guest"):
    print(f"Hello, {name}!")

# Call with an argument
greet("Alice") # Output: Hello, Alice!

# Call without an argument
greet() # Output: Hello, Guest!
```

```
Hello, Alice!
Hello, Guest!
```

Note: Default arguments must come after required arguments in the parameter list.

```
def greet(name="Guest", message):
    print(f"Hello, {name}! {message}")
```

SyntaxError: parameter without a default follows parameter with a default (3495907264.py, line 1)

```
Cell In[11], line 1
    def greet(name="Guest", message):
        ^
```

SyntaxError: parameter without a default follows parameter with a default

5.4.4.4 Variable-Length Arguments

These allow a function to accept an arbitrary number of arguments.

Using `*args` for Non-Keyword Variable-Length Arguments The `*args` syntax is used to pass a variable number of positional arguments. These arguments are accessible as a tuple.

```
def sum_numbers(*args):
    total = sum(args)
    print(f"The sum is {total}.")

# Call with multiple arguments
sum_numbers(1, 2, 3, 4)
sum_numbers(10, 20)

# Call without arguments
sum_numbers()
```

```
The sum is 10.
The sum is 30.
The sum is 0.
```

```
help(print)
```

Help on built-in function print in module builtins:

```
print(*args, sep=' ', end='\n', file=None, flush=False)
```

Prints the values to a stream, or to `sys.stdout` by default.

```
sep
    string inserted between values, default a space.
end
    string appended after the last value, default a newline.
file
    a file-like object (stream); defaults to the current sys.stdout.
flush
    whether to forcibly flush the stream.
```

The `print()` function in Python allows you to pass multiple arguments, separated by commas. When you do so, `print()` automatically converts each argument to a string (if it's not already a string) and joins them with a default separator, which is a **space**.

Example with multiple arguments:

```
print("Hello", "world!", 123, True)
```

```
Hello world! 123 True
```

You can change the default separator using the `sep` parameter.

```
print("Hello", "world!", 123, True, sep="***")
```

```
Hello***world!***123***True
```

You can change the default `end` parameter as well

```
print("Hello", "world!", sep="***", end=" :) ")
print("This is fun!")
```

```
Hello***world! :) This is fun!
```

Using ****kwargs** for Keyword Variable-Length Arguments

The ****kwargs** syntax is used to accept a variable number of keyword arguments. These arguments are accessible as a dictionary.

```
def print_details(**kwargs):
    for key, value in kwargs.items():
        print(f"{key}: {value}")

# Call with keyword arguments
print_details(name="Alice", age=30, city="Chicago")
```

```
name: Alice
age: 30
city: Chicago
```

Example: Combining `*args` and `**kwargs`

```
def mixed_function(a, *args, **kwargs):
    print(f"Fixed argument: {a}")
    print(f"Args: {args}")
    print(f"Kwargs: {kwargs}")

mixed_function(1, 2, 3, name="Alice", age=30)
```

```
Fixed argument: 1
Args: (2, 3)
Kwargs: {'name': 'Alice', 'age': 30}
```

5.4.4.5 Combining All Types of Arguments

You can use all these types of arguments in a single function, but they must follow a specific order:

- 1) Required arguments
- 2) Default arguments
- 3) `*args`
- 4) `**kwargs`

```
def display_info(name, age=18, *hobbies, **details):
    print(f"Name: {name}")
    print(f"Age: {age}")
    print(f"Hobbies: {' '.join(hobbies)}")
    for key, value in details.items():
        print(f"{key}: {value}")
```

```
# Call with all types of arguments
display_info(
    "Alice",
    25,
    "reading", "traveling",
    city="Chicago", job="Data Scientist"
)
```

```
Name: Alice
Age: 25
Hobbies: reading, traveling
city: Chicago
job: Data Scientist
```

5.4.5 Practice exercise 3

Write a function that prints prime numbers between two real numbers - **a** and **b**, where **a** and **b** are the parameters of the function. Call the function and check the output with **a = 60**, **b = 80**.

5.4.6 Functions that return objects

Until now, we saw functions that print text. However, the functions did not **return** any object. For example, the function `odd_even` prints whether the number is odd or even. However, we did not save this information. In future, we may need to use the information that whether the number was odd or even. Thus, typically, we return an object from the function definition, which consists of the information we may need in the future.

The example `odd_even` can be updated to return the text “odd” or “even” as shown below:

```
#This is an example of a function definition that has an argument with a default value, and 1
def odd_even(num=0):
    if num%2==0:
        return("Even")
    else:
        return("Odd")
```

The function above returns a string “Odd” or “Even”, depending on whether the number is odd or even. This result can be stored in a variable, which can be used later.

```
response=odd_even(3)
response
```

'Odd'

The variable `response` now refers to the object where the string “Odd” or “Even” is stored. Thus, the result of the computation is stored, and the variable can be used later on in the program. Note that the control flow exits the function as soon as the first `return` statement is executed.

5.4.7 Practice exercise 4: Create a Custom Calculator

Write a function `calculator` that performs basic arithmetic operations: addition, subtraction, multiplication, and division.

Steps:

1. Define a function `calculator` with three parameters: `a`, `b`, and `operation` (a string indicating the operation, e.g., `'add'`, `'subtract'`).
2. Use conditional statements to handle the operations.
3. Return the result of the operation.
4. Handle invalid operations gracefully.

Expected Output Examples:

```
print(calculator(10, 5, 'add'))      # Output: 15
print(calculator(10, 5, 'subtract')) # Output: 5
print(calculator(10, 5, 'multiply')) # Output: 50
print(calculator(10, 5, 'divide'))   # Output: 2.0
```

5.4.8 Bonus question: Calculator Function with Variable Number of Inputs

Update the `calculator` function to handle a variable number of inputs using the `*args`

Hints

- **Variable-Length Arguments (`*args`):** The `*args` parameter allows the function to accept an arbitrary number of arguments. These arguments are collected into a tuple, making them easy to iterate over.
- **Handling the Operator:** The operator parameter remains a keyword argument, defaulting to `+`. Supported operators are `+`, `-`, `*`, and `/`.

- **Logic:** Start with the first value in `args` (`result = args[0]`). Iterate through the rest of the numbers in `args[1:]` and apply the operator cumulatively.
- **Validation:** If fewer than two numbers are provided, return an error message. Handle division by zero with an additional check.

5.4.9 Practice exercise 5: Palindrome Checker

Write a function called `is_palindrome` that checks if a string is a palindrome (reads the same forward and backward).

Steps:

1. Define the function `is_palindrome` with one parameter, `text`.
2. Ignore case and spaces.
3. Return `True` if the string is a palindrome, otherwise `False`.

Expected OUPut Examples:

```
print(is_palindrome("radar"))           # Output: True
print(is_palindrome("hello"))          # Output: False
print(is_palindrome("A man a plan a canal Panama")) # Output: True
```

5.4.10 Global and local variables with respect to a function

5.4.10.1 Local Variables

- **Definition:** A local variable is defined within a function (or a block of code) and can only be accessed within that function. Once the function finishes executing, local variables are discarded.
- **Scope:** Limited to the function in which they are declared

```
def example_function():
    x = 10          # x is a local variable
    print("Inside function, x =", x)

example_function()
# Trying to print x outside the function will result in an error:
print(x) # This will result in an error
```

Inside function, x = 10

NameError: name 'x' is not defined

```
-----
NameError                                Traceback (most recent call last)
Cell In[19], line 7
      5 example_function()
      6 # Trying to print x outside the function will result in an error:
----> 7 print(x) # NameError: name 'x' is not defined
NameError: name 'x' is not defined
```

5.4.10.2 Global Variables

- **Definition:** A global variable is declared in the main body of the Python file (i.e., at the top level), making it accessible to any function or class in the same module, provided you do not shadow it with a local variable.
- **Scope:** Accessible throughout the entire module (file) after declaration.
- **Best Practice:** Use global variables sparingly, as they can make code harder to debug and maintain.

```
global_var = 20 # global variable

def show_global_var():
    print("Inside function, global_var =", global_var)

show_global_var()
print("Outside function, global_var =", global_var)
```

```
Inside function, global_var = 20
Outside function, global_var = 20
```

5.4.10.3 Using a Global Variable Inside a Function

By default, if you just read a global variable inside a function, Python will find it in the global scope. However, if you attempt to modify a global variable inside a function without explicitly declaring it `global`, Python will treat that variable as `local`, potentially leading to errors.

```
# Reading a global variable inside a function
global_var = 20 # global variable

def show_global_var():
    print("Inside function, global_var =", global_var)
```

```
show_global_var()
```

Inside function, `global_var = 20`

```
# Changing a global variable inside a function
counter = 0
```

```
def increment_counter_wrong():
    counter = counter + 1 # This will cause UnboundLocalError
    print("Counter is now", counter)
```

```
increment_counter_wrong() # UnboundLocalError: local variable 'counter' referenced before as
```

UnboundLocalError: cannot access local variable 'counter' where it is not associated with a v

UnboundLocalError Traceback (most recent call last)

Cell In[22], line 8

```
5     counter = counter + 1 # This will cause UnboundLocalError
6     print("Counter is now", counter)
```

----> 8 increment_counter_wrong() # UnboundLocalError: local variable 'counter' referenced

Cell In[22], line 5, in increment_counter_wrong()

```
4 def increment_counter_wrong():
----> 5     counter = counter + 1 # This will cause UnboundLocalError
6     print("Counter is now", counter)
```

UnboundLocalError: cannot access local variable 'counter' where it is not associated with a v

Why the Error? Python sees `counter = counter + 1` as creating a new **local variable** `counter` on the left, while also trying to read an **uninitialized** local variable `counter` on the right.

5.4.10.4 The global keyword

To modify a global variable inside a function, you must use the `global` keyword:

```
counter = 0 # global variable

def increment_counter():
    global counter # Tell Python we want to use the global 'counter'
    counter += 1
    print("Counter is now", counter)
```

```
increment_counter() # Counter is now 1
increment_counter() # Counter is now 2
print(counter)
```

```
Counter is now 1
Counter is now 2
2
```

When to Use `global`? * **Rarely.** Global variables can create tightly coupled code that is prone to bugs. If needed, consider passing variables as arguments or using class-level variables for shared state.

5.4.10.5 Nested Functions and the `nonlocal` Keyword

In Python, you may have **nested functions**—a function defined inside another function. If the inner function needs to modify a variable in the outer (but still non-global) scope, you can use the `nonlocal` keyword.

For example:

```
def outer_function():
    x = 10

    def inner_function():
        nonlocal x
        x += 5
        print("Inner function, x =", x)

    inner_function()
    print("Outer function, x =", x)

outer_function()
```

```
Inner function, x = 15
Outer function, x = 15
```

`nonlocal x` lets the inner function modify `x` in the `outer_function`'s scope, rather than creating a new local variable.

5.4.11 Practice exercise 6

Read the following code and answer the following questions:

1. Will the program raise an error?
2. If yes, fix the error and provide the corrected code.
3. What will be the output of the corrected program?

```
message = "Global Message" # Global variable

def outer():
    msg_outer = "Outer Message" # Enclosed scope

    def inner():
        message
        msg_outer

        message = "Changed Global Message"
        msg_outer = "Changed Outer Message"
        local_msg = "Local to inner()"
        print("Inside inner()")
        print("Global message =", message)
        print("Enclosed msg_outer =", msg_outer)
        print("Local local_msg =", local_msg)

    inner()
    print("\nInside outer() after inner() call:")
    print("Global message =", message)
    print("Enclosed msg_outer =", msg_outer)
    print("Local local_msg =", local_msg)

outer()

print("\nOutside all functions (global scope):")
print("Global message =", message)
print("Enclosed msg_outer =", msg_outer)
print("Local local_msg =", local_msg)
```

6 Data structures

<IPython.core.display.Image object>

In Chapter 2, we learned about primitive data types, each of which represents a single value.

In this chapter, we will explore **container data types**, also known as **data structures** or **collection data types** in Python. These data types allow us to store multiple primitive values, such as integers, booleans, and strings, as well as objects of different data types, all within a single structure.

String is one type of container data type, consisting of a sequence of characters. You already learned about strings in previous chapters. In this chapter, we focus on four main container data types in Python: **list**, **tuple**, **set**, and **dictionary**. Each differs in terms of **order** and **immutability**:

- **List**: Ordered and mutable (elements can be changed).
- **Tuple**: Ordered and immutable (elements cannot be changed once defined).
- **Set**: Unordered and mutable (elements can be added or removed, but duplicates are not allowed).
- **Dictionary**: Ordered (as of Python 3.7) and mutable, with key-value pairs for efficient lookups.

We will explore their characteristics, use cases, and differences in detail in this chapter.

6.1 Lists

A List in Python is an **ordered**, **mutable** (changeable) collection of items. Lists are one of the most versatile data structures in Python.

6.1.1 Creating a List

You can create a list by enclosing items in square brackets, separated by commas:

```
# Creating lists
empty_list = []
numbers = [1, 2, 3, 4, 5]
mixed = [42, "hello", True, 3.14]
```

6.1.2 Accessing Elements

Lists are ordered collections with unique indexes for each item. We can access/slice the items in the list using this index number. Python supports both positive and negative indexing, as shown below:

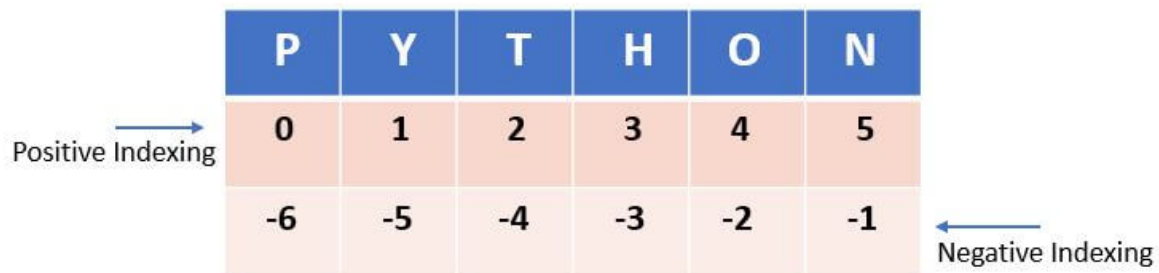


Figure 6.1: image.png

- **Indexing:** Use `[index]` to access a specific item.
- **Slicing:** Use `[start:end:step]` to get a sub-list.

```
my_list = [10, 20, 30, 40, 50]

print(my_list[0])      # 10
print(my_list[-1])     # 50
print(my_list[1:3])    # [20, 30]
print(my_list[2:])     # [30, 40, 50]
print(my_list[-3:-1])  # [30, 40]
print(my_list[::2])    # [10, 30, 50] (step of 2)
print(my_list[::-1])   # [50, 40, 30, 20, 10] (reverse)
```

```
10
50
[20, 30]
[30, 40, 50]
```

```
[30, 40]
[10, 30, 50]
[50, 40, 30, 20, 10]
```

6.1.3 Modifying a List

Lists can be changed after creation. You can add, remove, or replace elements.

6.1.3.1 Adding Items

- `append(item)`: Add item at the end.
- `insert(index, item)`: Insert item at a specific index.
- `extend(iterable)`: Extend the list by appending all items from an iterable (like another list).

```
fruits = ["apple", "banana"]
fruits.append("cherry")
print(fruits)

fruits.insert(1, "orange")
print(fruits)

fruits.extend(["grape", "mango"])
print(fruits)
```

```
['apple', 'banana', 'cherry']
['apple', 'orange', 'banana', 'cherry']
['apple', 'orange', 'banana', 'cherry', 'grape', 'mango']
```

Lists can be also concatenated using the `+` operator:

```
list_example = [5, 'hi', 4]
list_example = list_example + [None, '7', 9]
list_example
```

```
[5, 'hi', 4, None, '7', 9]
```

For adding elements to a list, the `extend` method is preferred over the `+` operator. This is because the `+` operator creates a new list, while the `extend` method adds elements to an existing list. Thus, the `extend` operator is more memory efficient.

6.1.3.2 Removing Items

- `pop([index])`: Removes and returns the item at index. If no index is given, removes the last item.
- `remove(value)`: Removes the first occurrence of value.
- `clear()`: Removes all items from the list, making it empty.

```
numbers = [10, 20, 30, 40, 50]
```

```
last_item = numbers.pop()
print(last_item)
print(numbers)
```

```
numbers.remove(20)
print(numbers)
```

```
numbers.clear()
print(numbers)
```

```
50
[10, 20, 30, 40]
[10, 30, 40]
[]
```

6.1.3.3 Replacing Items

You can directly reassign an element using the index:

```
letters = ["a", "b", "c", "d"]
letters[1] = "z"
print(letters)
```

```
['a', 'z', 'c', 'd']
```

6.1.3.4 Sorting a list

- `sort()`: Sorts the list in place.

```
nums = [4, 1, 3, 2]
nums.sort()
print(nums)
```

```
[1, 2, 3, 4]
```

```
help(list.sort)
```

Help on method_descriptor:

```
sort(self, /, *, key=None, reverse=False) unbound builtins.list method
    Sort the list in ascending order and return None.
```

The sort is in-place (i.e. the list itself is modified) and stable (i.e. the order of two equal elements is maintained).

If a key function is given, apply it once to each list item and sort them, ascending or descending, according to their function values.

The reverse flag can be set to sort in descending order.

By default, `list.sort()` sorts the list in ascending order. You can also pass a `reverse=True` argument to `sort()` to reverse the order:

```
nums.sort(reverse=True)
print(nums)
```

```
[4, 3, 2, 1]
```

You can also set the `key` parameter to a function that returns a value to sort by:

Example 1: Sorting strings by their length:

We define a function `get_length` that returns the length of a string. Then, we use this function as the `key` in the sort method.

```
def get_length(s):
    return len(s)

words = ["apple", "banana", "kiwi", "grape", "pineapple"]
words.sort(key=get_length) # Sorts by string length
print("Sorted by length:", words)
```

Sorted by length: ['kiwi', 'apple', 'grape', 'banana', 'pineapple']

Another way to achieve it. If a key function is given, apply it once to each list item and sort them, ascending or descending, according to their function values.

the `len` function will be applied to each list item.

```
words.sort(key=len, reverse=True) # Sorts by string length in reverse
print("Sorted by length:", words)
```

Sorted by length: ['pineapple', 'banana', 'apple', 'grape', 'kiwi']

Another way to define the key function is by using a `lambda` function. The concept of `lambda` functions will be covered in the later sequence course.

```
# use lambda function
words.sort(key=lambda s: s[-1]) # Sorts by last letter
print("Sorted by last letter:", words)
```

Sorted by last letter: ['banana', 'apple', 'grape', 'pineapple', 'kiwi']

Example 2: Sorting a List of Lists by a given Element

```
def sort_by_index(data, i):
    def get_element(lst):
        return lst[i] # Extract the element at index i

    data.sort(key=get_element) # Sort by the specified index

# Example usage:
data = [[1, 5, 9], [3, 2, 8], [4, 8, 6], [2, 6, 7]]

# Sort by the second element (index 1)
sort_by_index(data, 1)
print("Sorted by index 1:", data)

# Sort by the third element (index 2)
sort_by_index(data, 2)
print("Sorted by index 2:", data)
```

```
Sorted by index 1: [[3, 2, 8], [1, 5, 9], [2, 6, 7], [4, 8, 6]]
Sorted by index 2: [[4, 8, 6], [2, 6, 7], [3, 2, 8], [1, 5, 9]]
```

Let's use lambda function to rewrite it

```
data.sort(key=lambda lst: lst[0] + lst[1] + lst[2]) # Sort by sum of elements
print("Sorted by sum:", data)

# sort by the last element in descending order
data.sort(key=lambda lst: lst[-1], reverse=True)
print("Sorted by last element in descending order:", data)
```

```
Sorted by sum: [[3, 2, 8], [2, 6, 7], [1, 5, 9], [4, 8, 6]]
Sorted by last element in descending order: [[1, 5, 9], [3, 2, 8], [2, 6, 7], [4, 8, 6]]
```

6.1.3.5 Other Useful Methods

- `reverse()`: Reverses the list in place.
- `index(value)`: Returns the index of the first occurrence of value.

```
nums.reverse()
print(nums)

idx = nums.index(3)
print(idx)
```

```
[1, 2, 3, 4]
[4, 3, 2, 1]
1
```

```
help(list.reverse)
```

Help on method_descriptor:

```
reverse(self, /) unbound builtins.list method
  Reverse *IN PLACE*.
```

6.1.3.6 In-Place vs. Out-of-Place Operations

Note that both `list.sort()` and `list.reverse()` modify the list **in place**, meaning they do not create a new list but instead change the original one directly. To sort or reverse a list without modifying the original list in place, you can use functions that create a new list rather than updating the existing one:

Sorting Without In-Place Modification:

```
original_list = [3, 1, 2]
sorted_list = sorted(original_list) # returns a new, sorted list
print(original_list) # [3, 1, 2] (unchanged)
print(sorted_list)   # [1, 2, 3]
```

Reversing Without In-Place Modification:

```
# method 1: use the built-in reversed function
original_list = [3, 1, 2]
reversed_list = list(reversed(original_list))
print(original_list) # [3, 1, 2] (unchanged)
print(reversed_list) # [2, 1, 3]
```

```
[3, 1, 2]
[2, 1, 3]
```

```
# use list slicing (:: -1) to create a reversed copy of the original list
original_list = [3, 1, 2]
reversed_list = original_list[::-1]
print(original_list) # [3, 1, 2] (unchanged)
print(reversed_list) # [2, 1, 3]
```

```
[3, 1, 2]
[2, 1, 3]
```

6.1.4 Practice exercise 1

Start by defining a list that contains the elements [8, 9, 10]. Then do the following:

6.1.4.1

Set the second entry (index 1) to 17

6.1.4.2

Add 4, 5, and 6 to the end of the list

6.1.4.3

Remove the first entry from the list

6.1.4.4

Sort the list

6.1.4.5

Double the list (concatenate the list to itself)

6.1.4.6

Insert 25 at index 3, then print the final list. Expected Output: [4, 5, 6, 25, 10, 17, 4, 5, 6, 10, 17]

6.1.5 List Comprehension

List comprehension is a concise and elegant way to create lists in Python. It provides a shorter syntax to generate lists based on existing iterables while applying conditions or transformations.

Basic Syntax

```
[expression for item in iterable if condition]
```

Components:

- **Expression:** The value or transformation applied to each item.
- **for item in iterable:** Iterates over the iterable (e.g., list, range, string).
- **Condition (optional):** Filters items based on a condition.

Examples:

- Simple List Comprehension

```
# create a list of squares of numbers from 1 to 5
squares = [x**2 for x in range(1, 6)]
print(squares)
```

```
[1, 4, 9, 16, 25]
```

- List Comprehension with Condition

```
# create a list of even numbers from 1 to 10
evens = [x for x in range(1, 11) if x % 2 == 0]
print(evens) # Output: [2, 4, 6, 8, 10]
```

- List Comprehension with Transformation

```
# create a list of words with all letters in uppercase
words = ["hello", "world", "python"]
uppercase_words = [word.upper() for word in words]
print(uppercase_words)
```

```
['HELLO', 'WORLD', 'PYTHON']
```

- Nested List Comprehension

```
# flatten a 2D matrix into a 1D list
matrix = [[1, 2], [3, 4], [5, 6]]
flattened = [num for row in matrix for num in row]
print(flattened) # Output: [1, 2, 3, 4, 5, 6]
```

```
[1, 2, 3, 4, 5, 6]
```

Anything that can be accomplished using list comprehension can also be achieved with traditional Python loops. However, list comprehension often reduces the amount of code, making it more concise and readable. Additionally, it is often faster than equivalent for loops due to Python's optimized implementation.

Comparison with for loops

```
# using a for loop
squares = []
for x in range(1, 6):
    squares.append(x**2)
print(squares)
```

```
[1, 4, 9, 16, 25]
```

```
# using list comprehension

squares = [x**2 for x in range(1, 6)]
print(squares)
```

```
[1, 4, 9, 16, 25]
```

Both achieve the same result, but the list comprehension is more concise.

6.1.6 Practice exercise 2

Using list comprehension:

- Create a list of all odd numbers between 1 and 20.
- Generate a list of the lengths of each word in the list ["apple", "banana", "cherry"].
- Create a list of numbers divisible by both 3 and 5 from 1 to 100.

6.2 Tuples

A Tuple is an **ordered, immutable** (unchangeable) collection of items. Tuples can be thought of as lists that cannot be modified after creation.

6.2.1 Creating a Tuple

Use parentheses or the `tuple()` constructor.

```
my_tuple = (1, 2, 3)
another_tuple = "apple", "banana", "cherry" # Parentheses are optional
single_item_tuple = ("hello",)              # Note the trailing comma

converted_tuple = tuple([4, 5, 6])           # Using tuple() constructor
```

6.2.2 Accessing Elements

Indexing and slicing work similarly to lists:


```
t = (10, 20, 30, 40, 50)

print(t[0])    # 10
print(t[1:3])  # (20, 30)
```

```
10
(20, 30)
```

6.2.3 Immutability

Once you create a tuple, you **cannot modify** it:

```
t = (1, 2, 3)
t[0] = 10
```

```
TypeError: 'tuple' object does not support item assignment
```

```
-----
TypeError                                Traceback (most recent call last)
Cell In[21], line 2
      1 t = (1, 2, 3)
----> 2 t[0] = 10 # TypeError: 'tuple' object does not support item assignment
TypeError: 'tuple' object does not support item assignment
```

Tuple can be defined without the rounded brackets as well:

If you need to change elements, convert it to a list, modify the list, and convert back to a tuple (though this somewhat defeats the purpose of immutability).

6.2.4 Tuple methods

A couple of useful tuple methods are **count**, which counts the occurrences of an element in the tuple and **index**, which returns the position of the first occurrence of an element in the tuple:

```
tuple_example = (2,5,64,7,2,2)
```

```
tuple_example.count(2)
```

```
3
```

```
tuple_example.index(2)
```

0

6.2.5 Concatenating tuples

Tuples can be concatenated using the + operator to produce a longer tuple:

```
(2,7,4) + ("another", "tuple") + ("mixed","datatypes",5)
```

```
(2, 7, 4, 'another', 'tuple', 'mixed', 'datatypes', 5)
```

Multiplying a tuple by an integer results in repetition of the tuple:

```
(2,7,"hi") * 3
```

```
(2, 7, 'hi', 2, 7, 'hi', 2, 7, 'hi')
```

6.2.6 Why Use a Tuple?

A list seems to be much more flexible than tuple, and can replace a tuple almost everywhere. Then why use tuple at all?

Some of the advantages of a tuple over a list are as follows:

- **Data Integrity:** Tuples ensure the data cannot be modified accidentally.
- **Faster:** Tuples can be more memory efficient and faster to iterate over compared to lists.
- **Dictionary Keys:** Tuples can be used as keys in dictionaries (because they are immutable).

```
#Example showing tuples take less storage space than lists for the same elements
tuple_ex = (1, 2, 'Obama')
list_ex = [1, 2, 'Obama']
print("Space taken by tuple =",tuple_ex.__sizeof__()," bytes")
print("Space taken by list =",list_ex.__sizeof__()," bytes")
```

```
Space taken by tuple = 48 bytes
```

```
Space taken by list = 72 bytes
```

```
#Examples showing tuples takes lesser time to retrieve elements
import time as tm
tt = tm.time()
list_ex = list(range(1000000)) #List containinig whole numbers upto 1 million
a=(list_ex[::-2])
print("Time take to retrieve every 2nd element from a list = ", tm.time()-tt)

tt = tm.time()
tuple_ex = tuple(range(1000000)) #tuple containinig whole numbers upto 1 million
a=(tuple_ex[::-2])
print("Time take to retrieve every 2nd element from a tuple = ", tm.time()-tt)
```

```
Time take to retrieve every 2nd element from a list = 0.03211236000061035
Time take to retrieve every 2nd element from a tuple = 0.01900315284729004
```

```
tuple_example = 2, 7, 4
```

We can check the data type of a python object using the *type()* function. Let us check the data type of the object *tuple_example*.

```
type(tuple_example)
```

```
tuple
```

Elements of a tuple can be extracted using their index within square brackets. For example the second element of the tuple *tuple_example* can be extracted as follows:

```
tuple_example[1]
```

```
7
```

Note that an element of a tuple cannot be modified. For example, consider the following attempt in changing the second element of the tuple *tuple_example*.

```
tuple_example[1] = 8
```

```
TypeError: 'tuple' object does not support item assignment
```

```
-----  
TypeError                                Traceback (most recent call last)  
<ipython-input-6-6ceb38adde52> in <module>  
----> 1 tuple_example[1] = 8
```

```
TypeError: 'tuple' object does not support item assignment
```

The above code results in an error as tuple elements cannot be modified.

Now that we have an idea about tuple, let us try to think where it can be used.

```
<IPython.core.display.HTML object>
```

6.2.7 Tuple Comprehension?

There is no direct tuple comprehension in Python. However, Python does allow a similar construct that looks like tuple comprehension but actually creates a **generator expression**. If you want to create a tuple using comprehension-like syntax, you can explicitly convert the generator to a tuple.

```
gen = (x**2 for x in range(5))  
print(gen)
```

```
<generator object <genexpr> at 0x000001B5240C9BE0>
```

Here, `gen` is a generator, not a tuple. Generators are lazily evaluated, meaning values are computed on demand, making them memory-efficient.

To create a tuple, we can use the `tuple()` function to convert the generator into a tuple explicitly.

```
tup = tuple(x**2 for x in range(5))  
print(tup) # Output: (0, 1, 4, 9, 16)
```

```
(0, 1, 4, 9, 16)
```

Why No Direct Tuple Comprehension?

- Python uses parentheses for both tuple creation and generator expressions. To avoid ambiguity, Python reserves parentheses for generators in this context.
- Converting a generator to a tuple ensures explicit behavior and consistency.

Using List Comprehension for a List of Tuples:

In Python, list comprehension is often used to create a list of tuples because it combines the flexibility of tuple creation with the concise syntax of list comprehension.

Example: Create a list of tuples, where each tuple consists of a natural number and its square, for natural numbers ranging from 5 to 15.

```
sqrtnatural_no_5_15 = [(x,x**2) for x in range(5,16)]
print(sqrtnatural_no_5_15)
```

```
[(5, 25), (6, 36), (7, 49), (8, 64), (9, 81), (10, 100), (11, 121), (12, 144), (13, 169), (14, 196)]
```

6.2.8 Practice exercise 3

Below is a list consisting of responses to the question: “At what age do you think you will marry?” from students of the STAT303-1 Fall 2022 class.

```
exp_marriage_age=['24','30','28','29','30','27','26','28','30+', '26','28','30','30','30','pr
```

Use list comprehension to:

6.2.8.1

Remove the elements that are not integers - such as ‘*probably never*’, ‘30+’, etc. What is the length of the new list?

Hint: The built-in python function of the `str` class - `isdigit()` may be useful to check if the string contains only digits.

6.2.8.2

Cap the values greater than 80 to 80, in the clean list obtained in (1). What is the mean age when people expect to marry in the new list?

6.2.8.3

Determine the percentage of people who expect to marry at an age of 30 or more.

6.2.9 Practice exercise 4

USA's GDP per capita from 1960 to 2021 is given by the tuple T in the code cell below. The values are arranged in ascending order of the year, i.e., the first value is for 1960, the second value is for 1961, and so on. Print the years in which the GDP per capita of the US increased by more than 10%.

```
T = (3007, 3067, 3244, 3375, 3574, 3828, 4146, 4336, 4696, 5032, 5234, 5609, 6094, 6726, 7226, 7801)
```

- Determine the average GDP per capita for the given period (1960–2021).
- Identify the year with the highest GDP per capita.
- Identify the year with the lowest GDP per capita.
- Create a new tuple where each element represents the percentage change in GDP per capita compared to the previous year.
- Identify the year with the highest percentage increase in GDP per capita.
- Construct a tuple where each element is a pair in the format (year, GDP per capita), making it easier to analyze trends over time.

6.3 Sets

A set is a built-in data type in Python used to store **unordered, unique, and mutable** items. Sets are commonly used for operations like

- **Membership testing:** Quickly check if an item is in a set.
- **Eliminating duplicate entries:** Sets automatically ensure only unique elements are stored.
- **Mathematical set operations:** Supports union (`|`), intersection (`&`), and difference (`-`).

6.3.1 Creating a set

A set can be created using curly braces or the `set()` constructor

```
my_set = {1, 2, 3, 4}
print(my_set)

my_set = set([1, 2, 2, 3, 4])
print(my_set)
type(my_set)

my_empty_set=set()
print(my_empty_set)
```

```
{1, 2, 3, 4}
{1, 2, 3, 4}
set()
```

A set can be also created by removing repeated elements from lists.

```
my_list = [1,4,4,4,5,1,2,1,3]
my_set_from_list = list(set(my_list))
print(my_set_from_list)
```

```
[1, 2, 3, 4, 5]
```

6.3.2 Accessing Elements

Since sets are **unordered**, you cannot use indexing or slicing:

```
my_set = {"apple", "banana", "cherry"}
my_set[0]
```

```
TypeError: 'set' object is not subscriptable
```

```
-----
TypeError                                Traceback (most recent call last)
Cell In[60], line 2
      1 my_set = "apple", "banana", "cherry"
----> 2 my_set[0]
TypeError: 'set' object is not subscriptable
```

Instead, you typically check membership or iterate over all elements:

```
if "apple" in my_set:
    print("Apple is in the set")

for item in my_set:
    print(item)
```

```
Apple is in the set
cherry
banana
apple
```

6.3.3 Adding and Removing Items

- `add(item)`: Adds an item to the set (if it's not already present).
- `update(iterable)`: Adds multiple items (from another set, list, tuple, etc.).
- `remove(item)`: Removes the specified item (raises an error if not found).
- `discard(item)`: Removes the specified item (does not raise an error if not found).
- `pop()`: Removes and returns an arbitrary item from the set.
- `clear()`: Removes all items.

```
# Add an element to a set
print(my_set)
my_set.add(5)
print(my_set)

# Remove an element from a set
my_set.remove(3)
print(my_set)

# Remove an element that doesn't exist
my_set.remove(3) # This will raise a KeyError

# Remove an element that doesn't exist without raising an error
my_set.discard(3)
```

```
{1, 2, 4, 5}
{1, 2, 4, 5}
```


KeyError: 3

```
-----
KeyError                                Traceback (most recent call last)
Cell In[44], line 7
      4 print(my_set)
      6 # Remove an element from a set
----> 7 my_set.remove(3)
      8 print(my_set)
     10 # Remove an element that doesn't exist
KeyError: 3
```

Remove an element using `remove()` (raises an error if the element does not exist); Use `discard()` to remove an element (does not raise an error if the element does not exist):

6.3.4 Mathematical Set Operations

Sets are ideal for tasks involving unions, intersections, and differences. The table below explains these operations on sets

Operation	Symbol	Method	Description
Union		<code>set_a.union(set_b)</code>	Combines all unique elements from two sets.
Intersection	&	<code>set_a.intersection(set_b)</code>	Finds common elements between two sets.
Difference	-	<code>set_a.difference(set_b)</code>	Finds elements in <code>set_a</code> but not in <code>set_b</code> .
Symmetric Difference	^	<code>set_a.symmetric_difference(set_b)</code>	Finds elements in either set, but not both.

```
# Examples of Mathematical Operations on Sets
set_a = {1, 2, 3}
set_b = {3, 4, 5}

# Union
print(set_a | set_b)

# Intersection
print(set_a & set_b)

# Difference
print(set_a - set_b)
```

```
# Symmetric Difference
print(set_a ^ set_b)
```

```
{1, 2, 3, 4, 5}
{3}
{1, 2}
{1, 2, 4, 5}
```

6.3.5 Set Comprehension

We can do set comprehensions just like list comprehensions

```
# set comprehension
my_set = {x for x in 'hello'}
print(my_set)
type(my_set)
```

```
{'h', 'o', 'e', 'l'}
```

```
set
```

6.3.6 Practice exercise 5

The GDP per capita of USA for most years from 1960 to 2021 is given by the tuple `D_tuple` given in the code cell below.

```
D_tuple = ((1960, 3007), (1961, 3067), (1962, 3244), (1963, 3375), (1964, 3574), (1965, 3828),
(1966, 4146), (1967, 4336), (1968, 4696), (1970, 5234), (1971, 5609), (1972, 6094),
(1973, 6726), (1974, 7226), (1975, 7801), (1976, 8592), (1978, 10565), (1979, 11674),
(1980, 12575), (1981, 13976), (1982, 14434), (1983, 15544), (1984, 17121), (1985, 18237),
(1986, 19071), (1987, 20039), (1988, 21417), (1989, 22857), (1990, 23889), (1991, 24342),
(1992, 25419), (1993, 26387), (1994, 27695), (1995, 28691), (1996, 29968), (1997, 31459),
(1998, 32854), (2000, 36330), (2001, 37134), (2002, 37998), (2003, 39490), (2004, 41725),
(2005, 44123), (2006, 46302), (2007, 48050), (2008, 48570), (2009, 47195), (2010, 48651),
(2011, 50066), (2012, 51784), (2013, 53291), (2015, 56763), (2016, 57867), (2017, 59915),
(2018, 62805), (2019, 65095), (2020, 63028), (2021, 69288))
```

Determine which years between 1960 and 2021 are missing GDP per capita data.

6.4 Dictionary

A Dictionary in Python is an **mutable** collection of **key-value** pairs. It's used when you need to associate a specific value with a key and quickly access that value by using the key.

A dictionary in Python consists of key-value pairs, where both keys and values are Python objects. **While values can be of any data type, keys must be immutable objects**, such as strings, integers, or tuples. For example, a list can be used as a value in a dictionary, but not as a key, because lists are mutable and their elements can be changed.

Ordered Dictionaries: As of Python 3.7, the language specification guarantees that dictionaries maintain insertion order. This means you can reliably depend on the order in which keys were inserted when iterating over or converting the dictionary. Prior to Python 3.7, this behavior was not officially guaranteed (though in CPython 3.6, it happened to work that way in practice).

6.4.1 Creating a dictionary

Use braces `{}` or the `dict()` constructor.

```
# Using braces
person = {
    "name": "Alice",
    "age": 30,
    "city": "New York"
}
print(person)

# Using dict() constructor
car = dict(brand="Tesla", model="Model 3", year=2023)
print(car)

# Empty dictionary
empty_dict = {}
print(empty_dict)
```

```
{'name': 'Alice', 'age': 30, 'city': 'New York'}
{'brand': 'Tesla', 'model': 'Model 3', 'year': 2023}
{}
```

6.4.2 Accessing and Modifying Values

Access values by their keys using square bracket notation `[key]` or the `.get(key)` method.

```
person = {"name": "Alice", "age": 30, "city": "New York"}

# Access value
print(person["name"])
print(person.get("name"))

# Modify value
person["age"] = 31
print(person)

# Add new key-value pair
person["job"] = "Engineer"
print(person)
```

Alice

Alice

```
{'name': 'Alice', 'age': 31, 'city': 'New York'}
```

```
{'name': 'Alice', 'age': 31, 'city': 'New York', 'job': 'Engineer'}
```

6.4.3 Removing Keys

- `pop(key)`: Removes and returns the value for key.
- `del dictionary[key]`: Removes the key-value pair.
- `popitem()`: Removes and returns an arbitrary key-value pair (in Python 3.7+, it removes the last inserted item).
- `clear()`: Removes all items.

```
person = {"name": "Alice", "age": 30, "city": "New York"}

age = person.pop("age")
print(age)          # 30
print(person)       # {"name": "Alice", "city": "New York"}

del person["name"]
print(person)       # {"city": "New York"}

person.popitem()
```

```
print(person)      # {} (now empty)

person.clear()
print(person)      # {}
```

```
30
{'name': 'Alice', 'city': 'New York'}
{'city': 'New York'}
{}
{}

```

6.4.4 Iterating over elements of a dictionary

Use the following dictionary methods to retrieve all key and values at once:

- `keys()`: Returns the list of all keys present in the dictionary.
- `values()`: Returns the list of all values present in the dictionary
- `items()`: Returns all the items present in the dictionary. Each item will be inside a tuple as a key-value pair.

```
fruits = {"apple": 3, "banana": 5, "cherry": 2}
print(fruits.keys())
print(fruits.values())

for key,value in fruits.items():
    print("The Head of State of",key,"is",value)
```

```
dict_keys(['apple', 'banana', 'cherry'])
dict_values([3, 5, 2])
The Head of State of apple is 3
The Head of State of banana is 5
The Head of State of cherry is 2
```

6.4.5 Practice exercise 6

The GDP per capita of USA for most years from 1960 to 2021 is given by the dictionary D given in the code cell below.

Find:

1. The GDP per capita in 2015

- ```
D = {'1960':3007, '1961':3067, '1962':3244, '1963':3375, '1964':3574, '1965':3828, '1966':4146, '1967':4487, '1968':4841, '1969':5207, '1970':5584, '1971':5971, '1972':6368, '1973':6775, '1974':7191, '1975':7617, '1976':8052, '1977':8497, '1978':8951, '1979':9414, '1980':9886, '1981':10367, '1982':10857, '1983':11356, '1984':11864, '1985':12381, '1986':12907, '1987':13442, '1988':13986, '1989':14539, '1990':15091, '1991':15652, '1992':16222, '1993':16801, '1994':17389, '1995':17986, '1996':18592, '1997':19207, '1998':19831, '1999':20464, '2000':21106, '2001':21757, '2002':22417, '2003':23086, '2004':23764, '2005':24451, '2006':25147, '2007':25852, '2008':26566, '2009':27289, '2010':28021, '2011':28762, '2012':29512, '2013':30271, '2014':31039, '2015':31816, '2016':32602, '2017':33397, '2018':34201, '2019':35014, '2020':35836, '2021':36667, '2022':37507, '2023':38356, '2024':39214, '2025':40081, '2026':40957, '2027':41842, '2028':42736, '2029':43639, '2030':44551, '2031':45472, '2032':46402, '2033':47341, '2034':48289, '2035':49246, '2036':50212, '2037':51187, '2038':52171, '2039':53164, '2040':54166, '2041':55177, '2042':56197, '2043':57226, '2044':58264, '2045':59311, '2046':60367, '2047':61432, '2048':62506, '2049':63589, '2050':64681, '2051':65782, '2052':66892, '2053':68011, '2054':69139, '2055':70276, '2056':71422, '2057':72577, '2058':73741, '2059':74914, '2060':76096, '2061':77287, '2062':78487, '2063':79696, '2064':80914, '2065':82141, '2066':83377, '2067':84622, '2068':85876, '2069':87139, '2070':88411, '2071':89692, '2072':90982, '2073':92281, '2074':93589, '2075':94906, '2076':96232, '2077':97567, '2078':98911, '2079':100264, '2080':101626, '2081':102997, '2082':104377, '2083':105766, '2084':107164, '2085':108571, '2086':109987, '2087':111412, '2088':112846, '2089':114289, '2090':115741, '2091':117192, '2092':118652, '2093':120121, '2094':121599, '2095':123086, '2096':124582, '2097':126087, '2098':127601, '2099':129124, '2100':130656, '2101':132197, '2102':133747, '2103':135306, '2104':136874, '2105':138451, '2106':140037, '2107':141632, '2108':143236, '2109':144849, '2110':146471, '2111':148092, '2112':149722, '2113':151361, '2114':153009, '2115':154666, '2116':156332, '2117':158007, '2118':159691, '2119':161384, '2120':163086, '2121':164797, '2122':166517, '2123':168246, '2124':169984, '2125':171731, '2126':173487, '2127':175252, '2128':177026, '2129':178809, '2130':180601, '2131':182402, '2132':184212, '2133':186031, '2134':187859, '2135':189696, '2136':191542, '2137':193397, '2138':195261, '2139':197134, '2140':199016, '2141':200907, '2142':202807, '2143':204716, '2144':206634, '2145':208561, '2146':210497, '2147':212442, '2148':214396, '2149':216359, '2150':218331, '2151':220312, '2152':222302, '2153':224301, '2154':226309, '2155':228326, '2156':230352, '2157':232387, '2158':234431, '2159':236484, '2160':238546, '2161':240617, '2162':242697, '2163':244786, '2164':246884, '2165':248991, '2166':251107, '2167':253232, '2168':255366, '2169':257509, '2170':259661, '2171':261822, '2172':263992, '2173':266171, '2174':268359, '2175':270556, '2176':272762, '2177':274977, '2178':277201, '2179':279434, '2180':281676, '2181':283927, '2182':286187, '2183':288456, '2184':290734, '2185':293021, '2186':295317, '2187':297622, '2188':299936, '2189':302259, '2190':304591, '2191':306932, '2192':309282, '2193':311641, '2194':314009, '2195':316386, '2196':318772, '2197':321167, '2198':323571, '2199':325984, '2200':328406, '2201':330837, '2202':333277, '2203':335726, '2204':338184, '2205':340651, '2206':343127, '2207':345612, '2208':348106, '2209':350609, '2210':353131, '2211':355662, '2212':358202, '2213':360751, '2214':363309, '2215':365876, '2216':368452, '2217':371037, '2218':373631, '2219':376234, '2220':378846, '2221':381467, '2222':384097, '2223':386736, '2224':389384, '2225':392041, '2226':394707, '2227':397382, '2228':400066, '2229':402759, '2230':405461, '2231':408172, '2232':410892, '2233':413621, '2234':416359, '2235':419106, '2236':421862, '2237':424627, '2238':427401, '2239':430184, '2240':432976, '2241':435777, '2242':438587, '2243':441406, '2244':444234, '2245':447071, '2246':449917, '2247':452772, '2248':455636, '2249':458509, '2250':461391, '2251':464282, '2252':467182, '2253':470091, '2254':473009, '2255':475936, '2256':478872, '2257':481817, '2258':484771, '2259':487734, '2260':490706, '2261':493687, '2262':496677, '2263':499676, '2264':502684, '2265':505701, '2266':508727, '2267':511762, '2268':514806, '2269':517859, '2270':520921, '2271':523992, '2272':527072, '2273':530161, '2274':533259, '2275':536366, '2276':539482, '2277':542607, '2278':545741, '2279':548884, '2280':552036, '2281':555197, '2282':558367, '2283':561546, '2284':564734, '2285':56
```

```
print("GDP per capita in 2015 =", D['2015'])
D['2014'] = (D['2013']+D['2015'])/2

#Iterating over all years from 1960 to 2021
for i in range(1960,2021):

 #Imputing the GDP of the year if it is missing
 if str(i) not in D.keys():
 D[str(i)] = (D[str(i-1)]+D[str(i+1)])/2
 print("Imputed GDP per capita for the year",i,"is $",D[str(i)])
```

```
'value': 1}, {'starbucks_types_substitution': {'value': 10}, 'starbucks_ref_desks': 5, 'starbucks_ref_desks_severage_nutrition': 1, 'starbucks_ref_desks_severage_nutrition_type': 'c'}
```

126

#### 6.4.7.1

If the object in (1) is a dictionary, what is the datatype of the values of the dictionary?

#### 6.4.7.2

If the object in (1) is a dictionary, what is the datatype of the elements within the values of the dictionary?

#### 6.4.7.3

How many calories are there in `Iced Coffee`?

#### 6.4.7.4

Which drink(s) have the highest amount of protein in them, and what is that protein amount?

#### 6.4.7.5

Which drink(s) have a fat content of more than 10g, and what is their fat content?

## 6.5 Choosing the Right Data Structure

### 1. List

- *Ordered* and *Mutable*
- Best for collections of related items you need to modify, iterate, or reorder frequently.

### 2. Tuple

- *Ordered* and *Immutable*
- Ideal for data that shouldn't change or for use as dictionary keys (since they are hashable).

### 3. Dictionary

- *Unordered* (maintains insertion order in Python 3.7+) and *Mutable*

- Perfect for key-value lookups, fast data retrieval based on unique keys, and clearly organizing named data.

#### 4. Set

- *Unordered* and *Mutable*
- Automatically enforces *unique* elements. Great for membership testing (e.g., `in` checks) and set operations like union, intersection, etc.

## 6.6 Immutable and mutable Data Types in Python

In Python, **immutable data types** are those whose values cannot be changed after they are created. If you try to modify an immutable object, Python will create a new object instead of changing the original one.

Here is a list of immutable data types in python

- Integers (`int`)
- Floats (`float`)
- Booleans (`bool`)
- Strings (`str`)
- Tuples (`tuple`)

### Key Characteristics of Immutable Data Types

- **Cannot be modified in place:** Any operation that appears to modify an immutable object actually creates a new object.
- **Hashable:** Immutable objects can be used as keys in dictionaries or elements in sets because their values do not change.
- **Memory Efficiency:** Python may reuse memory for immutable objects (e.g., small integers or short strings) to optimize performance.

```
when you try to modify the interger, it creates a new integer object
my_integer = 32

Memory address of my_integer
print("Memory address of my_integers:", id(my_integer))

my_integer = 12
Memory address of my_integer
print("Memory address of my_integers:", id(my_integer))
```



Memory address of my\_integers: 140725600664984  
Memory address of my\_integers: 140725600664344

```
s = "hello"

s[0] = 'H'
```

TypeError: 'str' object does not support item assignment

```

TypeError Traceback (most recent call last)
Cell In[8], line 3
 1 s = "hello"
----> 3 s[0] = 'H'
TypeError: 'str' object does not support item assignment
```

```
Immutable example: Strings
s = "hello"

print("Original string:", s)
print("Memory address of original s:", id(s))

s = s.upper()
print("New string:", s.upper())
print("Memory address of s.upper:", id(s.upper))
```

Original string: hello  
Memory address of original s: 1341161965296  
New string: HELLO  
Memory address of s.upper: 1341172556928

For comparison, here are some **mutable data types** in Python:

- Lists (**list**)
- Dictionaries (**dict**)
- Sets (**set**)

```
Mutable example: Lists
lst = [1, 2, 3]
print("list before modification:", lst)
print("Memory address of lst:", id(lst))

Modifying the list
```

```
lst.append(4)
print("list after modification:", lst)
print("Memory address of lst:", id(lst))
```

```
list before modification: [1, 2, 3]
Memory address of lst: 1341172415488
list after modification: [1, 2, 3, 4]
Memory address of lst: 1341172415488
```

## 6.7 Final Thoughts

- **Lists** are your go-to when you need an adjustable sequence of ordered items.
- **Tuples** provide a way to store data in an immutable sequence, ensuring it remains unchanged.
- **Dictionaries** let you organize data into key-value pairs for quick lookups and clearer data structures.
- **Sets** focus on uniqueness and membership operations, which can greatly optimize tasks like deduplication and intersection.

Choose the right data structure based on your needs to write concise, efficient, and easy-to-maintain code.

### 6.7.1 Bonus Practice exercise

The object `deck` defined below corresponds to a deck of cards. Estimate the probability that a five card hand will be a [flush](#), as follows:

1. Write a function that accepts a hand of 5 cards as argument, and returns whether the hand is a flush or not.
2. Randomly pull a hand of 5 cards from the deck. Call the function developed in (1) to determine if the hand is a flush.
3. Repeat (2) 10,000 times.
4. Estimate the probability of the hand being a flush from the results of the 10,000 simulations.

You may use the function [shuffle\(\)](#) from the `random` library to shuffle the deck everytime before pulling a hand of 5 cards.

```
deck = [{'value':i, 'suit':c}
for c in ['spades', 'clubs', 'hearts', 'diamonds']
for i in range(2,15)]
```

**Solution:**

```
import random as rm

#Function to check if a 5-card hand is a flush
def chck_flush(hands):

 #Assuming that the hand is a flush, before checking the cards
 yes_flush = 1

 #Storing the suit of the first card in 'first_suit'
 first_suit = hands[0]['suit']

 #Iterating over the remaining 4 cards of the hand
 for j in range(1,len(hands)):

 #If the suit of any of the cards does not match the suit of the first card, the hand
 if first_suit!=hands[j]['suit']:
 yes_flush = 0;

 #As soon as a card with a different suit is found, the hand is not a flush and t
 break;
 return yes_flush

flush=0
for i in range(10000):

 #Shuffling the deck
 rm.shuffle(deck)

 #Picking out the first 5 cards of the deck as a hand and checking if they are a flush
 #If the hand is a flush it is counted
 flush=flush+chck_flush(deck[0:5])

print("Probability of obtaining a flush=", 100*(flush/10000),"%")
```

Probability of obtaining a flush= 0.26 %

## 6.8 Resources

- [Dictionary Comprehension](#)
- [List Comprehensions, Dictionary Comprehensions](#)

# 7 Python Iterables

In Python, an **iterable** is an object capable of returning its members one at a time. Common examples of iterables include strings, lists, tuples, sets, dictionaries, and Ranges. Iterables are fundamental in Python for loops, comprehensions, and many built-in functions.

## 7.1 What are Python Iterables

### 7.1.1 What Makes an Object Iterable?

An object is considered iterable if it implements the `__iter__()` method, which returns an iterator object, or the `__getitem__()` method, which allows it to be accessed sequentially.

```
my_list = [1, 2, 3, 4, 5]
help(my_list.__iter__)
```

Help on method-wrapper:

```
__iter__() unbound builtins.list method
 Implement iter(self).
```

```
help(my_list.__getitem__)
```

Help on built-in function `__getitem__`:

```
__getitem__(index, /) method of builtins.list instance
 Return self[index].
```

### 7.1.2 Checking if an Object is Iterable

You can check if an object is iterable using the `collections.abc.Iterable` class:

```
from collections.abc import Iterable

print(isinstance([1, 2, 3], Iterable))
print(isinstance(123, Iterable))
print(isinstance(range(5), Iterable))
print(isinstance("hello", Iterable))
```

True  
False  
True  
True

## 7.2 Common Iterables in Python

Here are some built-in iterables in Python:

### 7.2.1 Strings

```
for char in "Python":
 print(char, end=" ")
```

P y t h o n

### 7.2.2 Lists

```
for num in [1, 2, 3]:
 print(num, end=" ")
```

1 2 3

### 7.2.3 Tuples

```
for item in (4, 5, 6):
 print(item, end=" ")
```

4 5 6

### 7.2.4 Sets

```
for elem in {1, 2, 3}:
 print(elem, end=" ")
```

1 2 3

### 7.2.5 Dictionaries

```
my_dict = {"a": 1, "b": 2}
for key in my_dict:
 print(key, end=" ")
```

a b

### 7.2.6 Ranges

```
for i in range(5): # Generates numbers from 0 to 4
 print(i, end=" ")
```

0 1 2 3 4

## 7.3 Iterating over Iterables

You can iterate over an iterable using: \* A `for` loop. \* the `iter()` and `next()` functions

We've often used `for` loops to traverse iterables in Python. Now, let's dive into how you can manually iterate over an iterable using the built-in `iter()` and `next()` functions. This approach offers finer control over the iteration process and can be especially useful in more advanced scenarios.

```
Example: Manually Iterating Over an Iterable Using iter() and next()

Define an iterable, such as a list of numbers
numbers = [10, 20, 30, 40, 50]

Obtain an iterator from the iterable
iterator = iter(numbers)

Use a while loop to manually iterate over the elements
while True:
 try:
 # Retrieve the next element in the iterator
 number = next(iterator)
 print(number)
 except StopIteration:
 # When there are no more elements, a StopIteration exception is raised.
 print("Iteration complete!")
 break
```

```
10
20
30
40
50
Iteration complete!
```

## 7.4 Iterables Unpacking

Python supports unpacking for iterables, allowing you to assign elements of an iterable to variables in a concise and readable way. Unpacking is a powerful feature that works with python iterable objects.

### 7.4.1 Basic Unpacking

You can unpack the elements of an iterable into separate variables:

```
Unpacking a list
numbers = [1, 2, 3]
a, b, c = numbers
print(a, b, c) # Output: 1 2 3
```



1 2 3

**Note:** The number of variables must match the number of elements in the iterable, or Python will raise a `ValueError`.

### 7.4.2 Extended (\*) Unpacking

Using the `*` operator, you can unpack multiple elements into a single variable, capturing the remaining elements as a list:

```
numbers = [1, 2, 3, 4, 5]

a, *b, c = numbers
print(a)
print(b)
print(c)
```

```
1
[2, 3, 4]
5
```

Here, `b` (with `*`) captures all the middle elements as a list.

### 7.4.3 Unpacking with Functions

You can use unpacking to pass iterable elements as arguments to functions:

```
unpacking a list or tuple
def add(a, b, c):
 return a + b + c

numbers = [1, 2, 3]
result = add(*numbers)
print(result)
```

6

```
unpacking a dictionary
def add(a=0, b=0, c=0):
 return a + b + c

numbers = {"a": 1, "b": 2, "c": 3}
result = add(**numbers)
print(result)
```

6

If we are interested in retrieving only some values of the tuple, the expression `*_` can be used to discard the other values. Let's say we are interested in retrieving only the first and the last two values of the tuple:

```
x,_,y,z = (4.5, "this is a string", (("Nested tuple",5)), "99",99)
```

```
x, y , z
```

```
(4.5, '99', 99)
```

## 7.5 Built-in Functions for Iterables

Python provides a variety of built-in functions to operate on iterables, making it easy to manipulate, process, and analyze collections like lists, tuples, strings, sets, and dictionaries. Below is a list of commonly used built-in functions specifically designed for iterables.

### 7.5.1 General Functions

| Function              | Description                                                                   | Example                                    |
|-----------------------|-------------------------------------------------------------------------------|--------------------------------------------|
| <code>len()</code>    | Returns the number of elements in an iterable.                                | <code>len([1, 2, 3]) → 3</code>            |
| <code>min()</code>    | Returns the smallest element in an iterable.                                  | <code>min([3, 1, 4]) → 1</code>            |
| <code>max()</code>    | Returns the largest element in an iterable.                                   | <code>max([3, 1, 4]) → 4</code>            |
| <code>sum()</code>    | Returns the sum of elements in an iterable (numeric types only).              | <code>sum([1, 2, 3]) → 6</code>            |
| <code>sorted()</code> | Returns a sorted list from an iterable (does <b>not</b> modify the original). | <code>sorted([3, 1, 2]) → [1, 2, 3]</code> |

| Function                        | Description                                                                                                        | Example                                                                        |
|---------------------------------|--------------------------------------------------------------------------------------------------------------------|--------------------------------------------------------------------------------|
| <code>reversed()</code>         | Returns an iterator that accesses the elements of an iterable in reverse.                                          | <code>list(reversed([1, 2, 3])) → [3, 2, 1]</code>                             |
| <code>enumerate()</code>        | Returns an iterator of tuples containing indices and elements of the iterable.                                     | <code>list(enumerate(['a', 'b', 'c'])) → [(0, 'a'), (1, 'b'), (2, 'c')]</code> |
| <code>all()</code>              | Returns <b>True</b> if all elements of the iterable are true (or if empty).                                        | <code>all([True, 1, 'a']) → True</code>                                        |
| <code>any()</code>              | Returns <b>True</b> if <b>any</b> element of the iterable is true.                                                 | <code>any([False, 0, 'b']) → True</code>                                       |
| <code>str.join(Iterable)</code> | Joins the elements of an iterable (e.g., list, tuple) into a single string, using the given string as a separator. | <code>''.join(['a', 'b', 'c']) → 'abc'</code>                                  |

```
sorted(["apple", "orange", "banana"])
```

```
['apple', 'banana', 'orange']
```

```
max(["apple", "orange", "banana"])
```

```
'orange'
```

### 7.5.2 sorted()

The `sorted()` function in Python is used to return a new sorted list from an iterable. It is a versatile and powerful tool for sorting data in ascending or descending order, with the ability to customize sorting behavior using a key function.

```
numbers = [3, 1, 4, 1, 5]
sorted_numbers = sorted(numbers)
print(sorted_numbers) # Output: [1, 1, 3, 4, 5]
```

```
[1, 1, 3, 4, 5]
```

```
numbers = [3, 1, 4, 1, 5]
sorted_numbers = sorted(numbers, reverse=True)
print(sorted_numbers) # Output: [5, 4, 3, 1, 1]
```

```
[5, 4, 3, 1, 1]
```

```
word = "python"
sorted_chars = sorted(word)
print(sorted_chars) # Output: ['h', 'n', 'o', 'p', 't', 'y']
```

```
['h', 'n', 'o', 'p', 't', 'y']
```

The `key` parameter allows customization of sorting logic by applying a function to each element before comparison.

```
words = ["banana", "apple", "cherry"]
sorted_words = sorted(words, key=len)
print(sorted_words) # Output: ['apple', 'banana', 'cherry']
```

```
['apple', 'banana', 'cherry']
```

```
numbers = [-5, 3, -1, 7]
sorted_numbers = sorted(numbers, key=abs)
print(sorted_numbers) # Output: [-1, 3, -5, 7]
```

```
[-1, 3, -5, 7]
```

```
sorted(["apple", "orange", 32])
```

```
TypeError: '<' not supported between instances of 'int' and 'str'
```

```

TypeError Traceback (most recent call last)
Cell In[20], line 1
----> 1 sorted(["apple", "orange", 32])
TypeError: '<' not supported between instances of 'int' and 'str'
```

When you use functions like `max()`, `min()`, `sum()`, or `sorted` on an iterable, Python will try to perform comparisons or arithmetic operations between the elements. These functions work seamlessly when all elements are of types that can be compared or added together. However, if the iterable contains mixed types that are not inherently comparable (like strings and numbers), you'll run into errors.

**By ensuring the elements in your iterable are of compatible types, you can effectively use these functions without encountering runtime errors.**

### 7.5.2.1 Difference Between `sorted()` and `list.sort()`

| Feature             | <code>sorted()</code>                                   | <code>list.sort()</code>                             |
|---------------------|---------------------------------------------------------|------------------------------------------------------|
| <b>Return Value</b> | Returns a <b>new</b> sorted list                        | Modifies the list in place and returns <b>None</b> . |
| <b>Input Type</b>   | Works with any iterable (e.g., lists, tuples, strings). | Works <b>only</b> with lists.                        |
| <b>Usage</b>        | <code>sorted(iterable)</code>                           | <code>list.sort()</code>                             |

**Example:**

```
my_list = [3, 1, 2]

Using sorted()
new_list = sorted(my_list)
print("Original List:", my_list) # [3, 1, 2]
print("New Sorted List:", new_list) # [1, 2, 3]

Using list.sort()
my_list.sort()
print("List after list.sort():", my_list) # [1, 2, 3]
```

```
Original List: [3, 1, 2]
New Sorted List: [1, 2, 3]
List after list.sort(): [1, 2, 3]
```

### 7.5.3 `enumerate()`

The `enumerate()` function adds a counter to an iterable and returns it as an **enumerate object**, which can be iterated over to get both the **index** and the **value** of each element in the iterable.

#### 7.5.3.1 Syntax

```
enumerate(iterable, start=0)
```

```
fruits = ["apple", "banana", "cherry"]

for index, fruit in enumerate(fruits):
 print(f"Index: {index}, Fruit: {fruit}")
```

```
Index: 0, Fruit: apple
Index: 1, Fruit: banana
Index: 2, Fruit: cherry
```

```
change the start index
fruits = ["apple", "banana", "cherry"]

for index, fruit in enumerate(fruits, start=1):
 print(f"Index: {index}, Fruit: {fruit}")
```

```
Index: 1, Fruit: apple
Index: 2, Fruit: banana
Index: 3, Fruit: cherry
```

```
using enumerate() with a list comprehension
fruits = ["apple", "banana", "cherry"]
indexed_fruits = [(index, fruit) for index, fruit in enumerate(fruits, start=1)]
print(indexed_fruits)
```

```
[(1, 'apple'), (2, 'banana'), (3, 'cherry')]
```

```
Working with Strings
word = "python"

for index, char in enumerate(word):
 print(f"Index: {index}, Character: {char}")
```

```
Index: 0, Character: p
Index: 1, Character: y
Index: 2, Character: t
Index: 3, Character: h
Index: 4, Character: o
Index: 5, Character: n
```

### 7.5.4 zip()

The `zip()` function is a built-in Python function that combines two or more iterables (e.g., lists, tuples, strings) into a single iterator of tuples. It is commonly used to pair elements from multiple iterables based on their positions.

```
help(zip)
```

Help on class zip in module builtins:

```
class zip(object)
| zip(*iterables, strict=False) --> Yield tuples until an input is exhausted.
|
| >>> list(zip('abcdefg', range(3), range(4)))
| [('a', 0, 0), ('b', 1, 1), ('c', 2, 2)]
|
| The zip object yields n-length tuples, where n is the number of iterables
| passed as positional arguments to zip(). The i-th element in every tuple
| comes from the i-th iterable argument to zip(). This continues until the
| shortest argument is exhausted.
|
| If strict is true and one of the arguments is exhausted before the others,
| raise a ValueError.
|
| Methods defined here:
|
| __getattr__(self, name, /)
| Return getattr(self, name).
|
| __iter__(self, /)
| Implement iter(self).
|
| __next__(self, /)
| Implement next(self).
|
| __reduce__(...)
| Return state information for pickling.
|
| __setstate__(...)
| Set state information for unpickling.
|
| -----
```

```
| Static methods defined here:
|
| __new__(*args, **kwargs)
| Create and return a new object. See help(type) for accurate signature.
```

## Key Points

- **Aggregation of Iterables:**

`zip()` takes any number of iterables as arguments and returns an iterator of tuples, where each tuple contains the corresponding elements from all the iterables.

- **Length of the Result:**

The iterator stops when the shortest input iterable is exhausted. If the iterables have different lengths, elements from the longer iterables that do not have a corresponding element in the shorter ones are ignored.

- **Return Type:**

The object returned by `zip()` is an iterator in Python 3. To convert it to a list or tuple, you can use the `list()` or `tuple()` functions.

## Basic Usage Example

```
combining two lists
list1 = [1, 2, 3]
list2 = ['a', 'b', 'c']

zipped = zip(list1, list2)
zipped
```

```
<zip at 0x20e27f3cec0>
```

```
print(zipped)
print(type(zipped))

is zipped an iterator?
print(isinstance(zipped, Iterable))
```

```
<zip object at 0x0000020E27F3CEC0>
<class 'zip'>
True
```



The `zip()` function returns an iterator, not a list. When you print an iterator directly, it doesn't display the elements but rather the object's memory address. To see the elements, you can iterate over it using a loop or convert the iterator into a list (or tuple or dict).

```
iterate over the zip object
for item in zipped:
 print(item)
```

```
(1, 'a')
(2, 'b')
(3, 'c')
```

```
convert the zip object to a list
list1 = [1, 2, 3]
list2 = ['a', 'b', 'c']

zipped = zip(list1, list2)
print(list(zipped))
```

```
[(1, 'a'), (2, 'b'), (3, 'c')]
```

```
Creating dictionaries from two list
keys = ['name', 'age', 'city']
values = ['Alice', 30, 'New York']

dictionary = dict(zip(keys, values))
print(dictionary)
```

```
{'name': 'Alice', 'age': 30, 'city': 'New York'}
```

```
names = ['Alice', 'Bob', 'Charlie']
scores = [85, 90, 95]

for name, score in zip(names, scores):
 print(f"{name} scored {score}")
```

```
Alice scored 85
Bob scored 90
Charlie scored 95
```

### 7.5.5 unzipping

You can “unzip” a zipped object using the `zip(*zipped)` syntax:

```
zipped = [(1, 'a'), (2, 'b'), (3, 'c')]
unzipped = zip(*zipped)

list1, list2 = list(unzipped)
print(list1)
print(list2)
```

```
(1, 2, 3)
('a', 'b', 'c')
```

## 7.6 Practice exercises

### 7.6.1 Exercise 1: Iterating Using a For Loop and Manual Iteration

#### 1. Using a For Loop:

- Given the list:

```
numbers = [10, 20, 30, 40, 50]
```

Write a `for` loop to print each element.

#### 2. Using `iter()` and `next()`:

- Obtain an iterator from the list using `iter()`.
- Use a `while` loop and `next()` to print each element.
- Handle the `StopIteration` exception gracefully when there are no more elements.

*Hint:*

Remember that calling `next()` on an exhausted iterator raises a `StopIteration` exception.

### 7.6.2 Exercise 2: Combining Iterables with `zip()`

#### 1. Zipping Two Lists:

- Given the lists:

```
list1 = [1, 2, 3]
list2 = ['a', 'b', 'c']
```

- Use the `zip()` function to combine these lists.
- Convert the result into a list and print it.

## 2. Iterating Over the Zipped Object:

- Iterate over the zipped object using a `for` loop and print each tuple.

*Note:*

The `zip()` function returns an iterator, so converting it to a list will reveal its elements.

### 7.6.3 Exercise 3: Unzipping a List of Tuples

#### 1. Given Data:

- You have a list of tuples:

```
pairs = [('apple', 1), ('banana', 2), ('cherry', 3)]
```

#### 2. Unzipping:

- Use the unpacking operator `(*)` along with `zip()` to separate the list into two tuples:
  - One tuple for the fruit names.
  - One tuple for the corresponding numbers.
- Print both tuples.

*Hint:*

Use the syntax `names, numbers = zip(*pairs)`.

### 7.6.4 Exercise 4: Using Built-in Functions on Iterables

#### 1. Numeric List Operations:

- Create a list:

```
numbers = [5, 3, 8, 1, 9]
```

#### 2. Apply Built-in Functions:

- Use the `min()`, `max()`, and `sum()` functions on this list.
- Print the results of each function.

*Note:*

These functions require the elements of the iterable to be of compatible types (e.g., all numbers).

# 8 Object-Oriented Programming

<IPython.core.display.Image object>

Python is a versatile programming language that supports object-oriented programming (OOP). **In Python, everything is an object, and built-in data types are implemented as classes.**

## 8.1 Classes

### 8.1.1 Built-in Data Types in Python

In Python, every built-in data type is implemented as a class. This includes:

- int
- float
- str
- list
- tuple
- dict
- set
- NoneType

You can confirm this by using the `type()` function or checking an object's `__class__` attribute:

```
print(type(42))
print(type(3.14))
print(type("hello"))
print(type([1, 2, 3]))
print(type(None))
```

```
<class 'int'>
<class 'float'>
<class 'str'>
```

```
<class 'list'>
<class 'NoneType'>
```

```
Checking the __class__ attribute
x = 42
print(x.__class__)
```

```
<class 'int'>
```

### 8.1.2 Understanding Classes in Python

A **class** is a blueprint for creating objects. It defines the **attributes** (characteristics) and **methods** (behaviors) that its objects will have.

For example, consider a **Cat** class: - **Attributes**: Characteristics shared by all cats, such as `breed`, `fur_color`, and `age`. - **Methods**: Actions that a cat can perform, such as `meow()`, `run()`, or `sleep()`.

For more details, refer to the official Python documentation on [classes](#).

### 8.1.3 Creating your own classes

Until now, we have worked with built-in Python classes like `int`, `list`, and `dict`. However, in many cases, we need to **create our own classes** to model real-world entities in a structured way.

Defining your own classes provides several key benefits in programming:

1. **Encapsulation** – Organize data and related functionality together.
2. **Reusability** – Code can be reused by creating multiple instances of the class.
3. **Abstraction** – Hide unnecessary details and expose only the required functionality.
4. **Inheritance** – Reuse existing class behavior in new classes, avoiding redundancy.

When we create a new class, we actually create a new type. Now, we are going to create our own type, which we can use in a way that is similar to the built-in types.

Let's start with the `Car` class:

```

class Car:
 def __init__(self, brand, color, speed=0):
 self.brand = brand # Attribute
 self.color = color # Attribute
 self.speed = speed # Attribute

 def accelerate(self, increment):
 """Increase the car's speed."""
 self.speed += increment
 return f"{self.brand} is now moving at {self.speed} mph."

 def brake(self, decrement):
 """Decrease the car's speed."""
 self.speed = max(0, self.speed - decrement)
 return f"{self.brand} slowed down to {self.speed} mph."

```

We'll use the example above to explain the following terms:

- **The class statement:** We use the class statement to create a class. The [Python style guide](#) recommends to use CamelCase for class names.
- **The constructor (or the `__init__()` method):** A class typically has a method called `__init__`. This method is called a constructor and is automatically called when an object or instance of the class is created. The constructor initializes the attributes of the class. In the above example, the constructor accepts Three values as arguments, and initializes its attributes `brand` and `color` with those values and have a default value for `speed`.
- **The `self` argument:** This is the first parameter of instance methods in a class. It represents the instance of the class itself, allowing access to its attributes and methods. When referring to instance attributes or methods within the class, they must be prefixed with `self`. The purpose of `self` is to distinguish instance-specific attributes and methods from local variables or other functions in the program.

This example demonstrates how a class encapsulates attributes and behaviors. The `Car` class defines three attributes: `brand`, `color`, and `speed`, along with a constructor (`__init__()`) and two methods: `accelerate()` and `brake()`. This structure makes it easy to create multiple car objects and manipulate their states independently.

## 8.2 Objects

In Python, an **object** is an instance of a class. A class acts as a **blueprint**, defining the structure and behavior that its objects will have. Each object has its own attributes (data) and can perform methods (functions associated with the class). Compared to the class (which is just a blueprint), an object is a **concrete and tangible** entity that exists in memory.

In Python, when you create a variable and assign a value to it, Python internally creates an instance of the corresponding class. Every value in Python is an object,

```
x = 10 # x is an instance of int class
y = "Hello" # y is an instance of str class
z = [1, 2, 3] # z is an instance of list class

print(type(x))
print(type(y))
print(type(z))
```

```
<class 'int'>
<class 'str'>
<class 'list'>
```

Once we define a class as a blueprint, we can create instances of that class to generate objects of its type. In fact, we can create as many objects as we want from a single class, each with its own unique data while sharing the same structure and behavior defined in the class.

Let's create two objects of the `Car` class we defined earlier

To create an *object* or *instance* of the class `Car`, we'll use the class name with the values to be passed as argument to the constructor for initializing the *object* / *instance*.

```
Creating objects (instances of the Car class)
car1 = Car("Toyota", "Red")
car2 = Car("Honda", "Blue")

print(type(car1))
print(type(car2))
```

```
<class '__main__.Car'>
<class '__main__.Car'>
```

**Instance:** An *instance* is a specific realization of the object of a particular class. Creating an *instance* of a class is called **Instantiation**. Here a particular car is an *instance* of the class `Car`. Similarly, in the example above, the object `x` is an instance of the class *integer*. **The words *object* and *instance* are often used interchangeably.**

The *attributes* of an instance can be accessed using the `.` operator with the object name

```
print(car1.brand)
print(car2.brand)
print(car1.color)
print(car2.color)
print(car1.speed)
print(car2.speed)
```

```
Toyota
Honda
Red
Blue
0
0
```

What happens if the instance variable doesn't exist?

```
car1.engine
```

```
AttributeError: 'Car' object has no attribute 'engine'
```

```

AttributeError Traceback (most recent call last)
Cell In[8], line 1
----> 1 car1.engine
AttributeError: 'Car' object has no attribute 'engine'
```

Methods are functions inside a class that operate on instance attributes. To call a method use:

```
print(car1.accelerate(20))
print(car2.brake(10))
```

```
Toyota is now moving at 20 mph.
Honda slowed down to 0 mph.
```



Unlike attributes, methods require **parentheses** () because they need to be executed, just like the functions we learned earlier

A list of all attributes and methods associated with an object can be obtained with the `dir()` function. Ignore the ones with underscores - these are used by Python itself. The rest of them can be used to perform operations.

```
dir(car1)
```

```
['__class__',
 '__delattr__',
 '__dict__',
 '__dir__',
 '__doc__',
 '__eq__',
 '__format__',
 '__ge__',
 '__getattr__',
 '__getstate__',
 '__gt__',
 '__hash__',
 '__init__',
 '__init_subclass__',
 '__le__',
 '__lt__',
 '__module__',
 '__ne__',
 '__new__',
 '__reduce__',
 '__reduce_ex__',
 '__repr__',
 '__setattr__',
 '__sizeof__',
 '__str__',
 '__subclasshook__',
 '__weakref__',
 'accelerate',
 'brake',
 'brand',
 'color',
 'speed']
```

Filtering out only user-defined attributes and methods

```
print([attr for attr in dir(car1) if not attr.startswith('__')])
```

```
['accelerate', 'brake', 'brand', 'color', 'speed']
```

### 8.2.1 Example: A class that analyzes a string

Let us create a class that analyzes a string.

```
class AnalyzeString:

 #Constructor
 def __init__(self, s):
 s = s.lower()
 self.words = s.split()

 #This method counts the numebr of words
 def number_of_words(self):
 return (len(self.words))

 #This method counts the number of words starting with the string s
 def starts_with(self,s):
 return len([x for x in self.words if x[:len(s)]==s])

 #This method counts the number of words of length n
 def words_with_length(self,n):
 return len([x for x in self.words if len(x)==n])

 #This method returns the frequency of the word w
 def word_frequency(self,w):
 return self.words.count(w)
```

Let us create an instance of the class `AnalyzeString()` to analyze a sentence.

```
#Defining a string
sentence = 'This sentence in an example of a string that we will analyse using a class we have'
```

```
#Creating an instance of class AnalyzeString()
sentence_analysis = AnalyzeString(sentence)
```

```
#The attribute 'word' contains the list of words in the sentence
sentence_analysis.words
```

```
['this',
 'sentence',
 'in',
 'an',
 'example',
 'of',
 'a',
 'string',
 'that',
 'we',
 'will',
 'analyse',
 'using',
 'a',
 'class',
 'we',
 'have',
 'defined']
```

```
#The method 'word_frequency()' provides the frequency of a word in the sentence
sentence_analysis.word_frequency('we')
```

2

```
#The method 'starts_with()' provides the frequency of number of words starting with a partic
sentence_analysis.starts_with('th')
```

2

### 8.2.2 Practice exercise 1

Write a class called `PasswordManager`. The class should have a list called `old_passwords` that holds all of the user's past passwords. The last item of the list is the user's current password. There should be a method called `get_password` that returns the current password and a method called `set_password` that sets the user's password. The `set_password` method should only change the password if the attempted password is different from all the user's past

passwords. It should either print *'Password changed successfully!'*, or *'Old password cannot be reused, try again.'* Finally, create a method called `is_correct` that receives a string and returns a boolean `True` or `False` depending on whether the string is equal to the current password or not.

To initialize the object of the class, use the list below.

After defining the class:

1. Check the attribute `old_passwords`
2. Check the method `get_password()`
3. Try re-setting the password to `'ibiza1972'`, and then check the current password.
4. Try re-setting the password to `'oktoberfest2022'`, and then check the current password.
5. Check the `is_correct()` method

```
class PasswordManager:
 def __init__(self, initial_passwords):
 self.old_passwords = initial_passwords

 def get_password(self):
 """Returns the current password (last item in the old_passwords list)."""
 return self.old_passwords[-1]

 def set_password(self, new_password):
 """Sets a new password only if it has not been used before."""
 if new_password in self.old_passwords:
 print("Old password cannot be reused, try again.")
 else:
 self.old_passwords.append(new_password)
 print("Password changed successfully!")

 def is_correct(self, password):
 """Checks if the provided password matches the current password."""
 return password == self.get_password()

Initialize PasswordManager with given passwords
initial_passwords = ["alpha123", "beta456", "gamma789", "delta321", "ibiza1972"]
password_manager = PasswordManager(initial_passwords)

1. Check the attribute old_passwords
print("Old passwords:", password_manager.old_passwords)
```

```

2. Check the method get_password()
print("Current password:", password_manager.get_password())

3. Try re-setting the password to 'ibiza1972'
password_manager.set_password("ibiza1972")
print("Current password after attempt:", password_manager.get_password())

4. Try re-setting the password to 'oktoberfest2022'
password_manager.set_password("oktoberfest2022")
print("Current password after attempt:", password_manager.get_password())

5. Check the is_correct() method
print("Is 'oktoberfest2022' correct?", password_manager.is_correct("oktoberfest2022"))
print("Is 'wrongpassword' correct?", password_manager.is_correct("wrongpassword"))

```

```

Old passwords: ['alpha123', 'beta456', 'gamma789', 'delta321', 'ibiza1972']
Current password: ibiza1972
Old password cannot be reused, try again.
Current password after attempt: ibiza1972
Password changed successfully!
Current password after attempt: oktoberfest2022
Is 'oktoberfest2022' correct? True
Is 'wrongpassword' correct? False

```

## 8.3 Class Constructors

A **constructor** is a special method in a class that is **automatically called** when an object is created.

- In Python, the constructor method is named `__init__()`.
- It **initializes object attributes** when an instance is created.

```

class Car:
 def __init__(self, brand, color, speed=0):
 """Constructor to initialize Car attributes"""
 self.brand = brand
 self.color = color
 self.speed = speed # Default value is 0

Creating instances (objects)
car1 = Car("Toyota", "Red", 50)

```

```
car2 = Car("Honda", "Blue") # speed uses default value

Accessing attributes
print(car1.brand, car1.color, car1.speed) # Toyota Red 50
print(car2.brand, car2.color, car2.speed)
```

```
Toyota Red 50
Honda Blue 0
```

### 8.3.1 Default Constructor (No Parameters)

If a class does not explicitly define a constructor, Python automatically provides a default constructor. This constructor only includes self and takes no additional parameters.

Let's create an empty class to demonstrate this:

```
Define a class Circle that doesn't have any attributes or methods
class Circle:
 pass

Create an instance of the Circle class
c = Circle()
print(type(c))
```

```
<class '__main__.Circle'>
```

### 8.3.2 Using Default Values in Constructors

You can set default values for parameters to make them optional.

```
class Student:
 def __init__(self, name, grade="Not Assigned"):
 self.name = name
 self.grade = grade # Default: "Not Assigned"

s1 = Student("John", "A") # Assigned grade
s2 = Student("Emma") # Uses default grade

print(s1.name, s1.grade) # John A
print(s2.name, s2.grade) # Emma Not Assigned
```

John A  
Emma Not Assigned

## 8.4 Difference Between Instance Attributes and Class Attributes in Python (OOP)

In Python object-oriented programming, attributes can be defined at two levels:

- **Instance Attributes** → Specific to each instance of the class.
- **Class Attributes** → Shared across all instances of the class.

### 8.4.1 Instance Attributes (Defined in `__init__`)

- Defined inside the constructor (`__init__`) using `self`, and accessed using `self.attribute_name`
- Each instance has its own copy of instance attributes.
- Changes to an instance attribute affect only that instance.

```
Example: Instance attributes using our Car class

class Car:
 def __init__(self, brand, color):
 self.brand = brand # Instance attribute
 self.color = color # Instance attribute

Creating instances
car1 = Car("Toyota", "Red")
car2 = Car("Honda", "Blue")

Each instance has different values
print(car1.brand) # Toyota
print(car2.brand) # Honda

Changing an instance attribute only affects that instance
car1.color = "Black"
print(car1.color) # Black
print(car2.color)
```

Toyota  
Honda  
Black  
Blue

### 8.4.2 Class Attributes ( Defined Outside `__init__`)

- Defined at the class level (outside `__init__`).
- Shared across all instances of the class.
- Changing a class attribute affects all instances (unless overridden at the instance level).

```
class Car:
 wheels = 4 # Class attribute (shared by all instances)

 def __init__(self, brand):
 self.brand = brand # Instance attribute

Creating instances
car1 = Car("Toyota")
car2 = Car("Honda")

Accessing class attribute
print(car1.wheels) # 4
print(car2.wheels) # 4

Changing the class attribute affects all instances
Car.wheels = 6
print(car1.wheels) # 6
print(car2.wheels) # 6
```

```
4
4
6
6
```

**Note:** Just like attributes, methods can be categorized into instance methods and class methods. So far, everything we have defined are instance methods. Class methods, however, are beyond the scope of this data science course and will not be covered.

## 8.5 Inheritance in Python

### 8.5.1 What is Inheritance?

**Inheritance** is a fundamental concept in Object-Oriented Programming (OOP) that allows a **child class** to inherit attributes and methods from a **parent class**. This promotes **code reuse** and **hierarchical structuring** of classes.



Here are Key Benefits of Inheritance:

- **Code Reusability** – Avoids redundant code by reusing existing functionality.
- **Extensibility** – Allows adding new functionality without modifying the original class.
- **Improves Maintainability** – Easier to manage and update related classes.

### 8.5.2 Defining a Parent (Base) and Child (Derived) Class

A child class **inherits** from a parent class by specifying the parent class name in parentheses.

**Example: Basic Inheritance**

```
Parent Class (Base Class)
class Animal:
 def __init__(self, name):
 self.name = name

 def speak(self):
 return "Some sound"

Child Class (Derived Class)
class Dog(Animal): # Inheriting from Animal
 def speak(self):
 return "Woof!"

Creating objects
dog1 = Dog("Buddy")

Accessing inherited attributes and methods
print(dog1.name)
print(dog1.speak())
```

Buddy  
Woof!

**Explanation:**

- Dog inherits from **Animal**, meaning it gets all the properties of **Animal**.
- The Dog class overrides the **speak()** method to provide a specialized behavior.

### 8.5.3 The super() Function

The `super()` function allows calling methods from the parent class inside the child class.

Example: Using `super()` to Extend Parent Behavior

```
class Animal:
 def __init__(self, name):
 self.name = name

 def speak(self):
 return "Some sound"

class Cat(Animal):
 def __init__(self, name, color):
 super().__init__(name) # Calling Parent Constructor
 self.color = color # Additional attribute in Child Class

 def speak(self):
 return "Meow!"

Creating an instance
cat1 = Cat("Whiskers", "Gray")

print(cat1.name) # Whiskers (Inherited from Animal)
print(cat1.color) # Gray (Defined in Cat)
print(cat1.speak()) # Meow! (Overridden method)
```

```
Whiskers
Gray
Meow!
```

- `super().__init__(name)` ensures the parent class constructor is properly called.
- This allows the child class to initialize both inherited and new attributes.

### 8.5.4 Method Overriding in Inheritance

- If a method exists in both the parent and child class, the child class's method overrides the parent's method.
- This is useful for customizing behavior.

Example: Overriding a Method

```
class Parent:
 def show(self):
 return "This is the Parent class"

class Child(Parent):
 def show(self): # Overriding method
 return "This is the Child class"

obj = Child()
print(obj.show()) # This is the Child class
```

This is the Child class

### 8.5.5 Check Relationship

- `issubclass(Child, Parent)` → Checks if a class is a subclass of another.
- `isinstance(object, Class)` → Checks if an object is an instance of a class.

```
class Animal:
 pass

class Dog(Animal):
 pass

dog1 = Dog()

print(issubclass(Dog, Animal)) # True
print(isinstance(dog1, Dog)) # True
print(isinstance(dog1, Animal)) # True (Since Dog inherits from Animal)
```

True  
True  
True

### 8.5.6 The object class in Python

When you define a class in Python without explicitly specifying a parent class, Python automatically makes it inherit from the built-in `object` class.

Let's confirm this using the `issubclass()` method

```
print(issubclass(Car, object))
```

True

You can use the `__bases__` attribute to check the parent class(es):

```
print(Car.__bases__)
```

(<class 'object'>,)

#### 8.5.6.1 What is the object class?

- `object` is the **base class for all classes in Python**.
- It provides default methods like:
  - `str()`
  - `repr()`
  - `eq()`
  - `init()`
  - And more...

Let's back to the whole list of `car1`

```
dir(car1)
```

```
['__class__',
'__delattr__',
'__dict__',
'__dir__',
'__doc__',
'__eq__',
'__format__',
'__ge__',
'__getattribute__',
```

```

'__getstate__',
'__gt__',
'__hash__',
'__init__',
'__init_subclass__',
'__le__',
'__lt__',
'__module__',
'__ne__',
'__new__',
'__reduce__',
'__reduce_ex__',
'__repr__',
'__setattr__',
'__sizeof__',
'__str__',
'__subclasshook__',
'__weakref__',
'brand',
'wheels']

```

Many special methods and attributes that start with double underscores (`__`) are inherited from the `object` class in Python. These are known as **dunder (double underscore) methods** or **magic methods**, such as `__init__()`, `__str__()`, and `__eq__()`. These methods are automatically called by Python for specific operations, and users typically do not need to call them directly.

We generally do not recommend modifying these methods unless you are customizing class behavior (e.g., overloading operators). If you need to define **private attributes or methods**, use a single underscore `_` (convention) or double underscore `__` (name mangling) to prevent accidental access.

### Key Takeaways:

- Inheritance promotes code reuse and hierarchy in OOP.
- A child class can inherit and override parent class methods.
- Use `super()` to call parent class methods inside the child class.

For more details, refer to the official Python documentation on [Inheritance](#).

## 8.5.7 Practice exercise 2

Define a class that inherits the in-built Python class `list`, and adds a new method to the class called `nunique()` which returns the number of unique elements in the list.

Define the following list as an object of the class you created. Then:

1. Find the number of unique elements in the object using the method `nunique()` of the inherited class.
2. Check if the `pop()` method of the parent class works to pop an element out of the object.

```
list_ex = [1,2,5,3,6,5,5,5,12]
```

```
class list_v2(list):
 def nunique(self):
 unique_elements = []
 for x in self:
 if x not in unique_elements:
 unique_elements.append(x)
 return len(unique_elements)

list_ex = list_v2(list_ex)
print("Number of unique elements = ", list_ex.nunique())
print("Checking the pop() method, the popped out element is", list_ex.pop())
```

```
Number of unique elements = 6
```

```
Checking the pop() method, the popped out element is 12
```

### 8.5.8 Practice exercise 3

Define a class named `PasswordManagerUpdated` that inherits the class `PasswordManager` defined in Practice exercise 1. The class `PasswordManagerUpdated` should have two methods, other than the *constructor*:

1. The method `set_password()` that sets a new password. The new password must only be accepted if it does not have any punctuations in it, and if it is not the same as one of the old passwords. If the new password is not acceptable, then one of the appropriate messages should be printed - (a) *Cannot have punctuation in password, try again*, or (b) *Old password cannot be reused, try again*.
2. The method `suggest_password()` that randomly sets and returns a password as a string comprising of 15 randomly chosen letters. Letters may be repeated as well.

```

import random
import string

class PasswordManagerUpdated(PasswordManager):
 def __init__(self, initial_passwords):
 super().__init__(initial_passwords)

 def set_password(self, new_password):
 """Sets a new password if it does not contain punctuation and is not a reused password"""
 if any(char in string.punctuation for char in new_password):
 print("Cannot have punctuation in password, try again.")
 elif new_password in self.old_passwords:
 print("Old password cannot be reused, try again.")
 else:
 self.old_passwords.append(new_password)
 print("Password changed successfully!")

 def suggest_password(self):
 """Generates and returns a random 15-character password without punctuation."""
 suggested_password = ''.join(random.choices(string.ascii_letters, k=15))
 self.old_passwords.append(suggested_password)
 return suggested_password

```

Note that `ascii_letters` constant in Python is part of the `string` module and is a predefined string that contains all the lowercase and uppercase ASCII letters. It contains all lowercase ('a' to 'z') and uppercase ('A' to 'Z') ASCII characters in sequence.

```

import string

Accessing ascii_letters
s = string.ascii_letters
print(s)

```

abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ

Let's use the class below:

```

Initialize PasswordManagerUpdated with given passwords
initial_passwords = ["alpha123", "beta456", "gamma789", "delta321", "ibiza1972"]
password_manager_updated = PasswordManagerUpdated(initial_passwords)

```

```
1. Try setting a password with punctuation
password_manager_updated.set_password("newPass@word!")

2. Try setting an already used password
password_manager_updated.set_password("ibiza1972")

3. Set a new valid password
password_manager_updated.set_password("securePass123")
print("Current password:", password_manager_updated.get_password())

4. Generate and set a suggested password
suggested = password_manager_updated.suggest_password()
print("Suggested password:", suggested)
print("Current password after suggestion:", password_manager_updated.get_password())
```

Cannot have punctuation in password, try again.  
Old password cannot be reused, try again.  
Password changed successfully!  
Current password: securePass123  
Suggested password: xXgtHYeOAPsVvvY  
Current password after suggestion: xXgtHYeOAPsVvvY



## **Part III**

### **R**

## 9 R: Basics



### 9.1 Comments

Comments in R start with the `#` symbol. Everything after `#` on a line is ignored by R.

```
This is a single-line comment
x <- 10 # Assigning 10 to x
```

R does not support multi-line comments like Python's `"""`, but you can simulate them using multiple `#` symbols:

```
This is a
multi-line comment
```

To comment a block of code quickly in RStudio, use:

- Ctrl + Shift + C (Windows/Linux)
- Cmd + Shift + C (Mac)

## 9.2 Data Types

R has several built-in data types that are crucial for data analysis and computation. The commonly used data types include:

- **Numeric** (double and integer)
- **Character** (strings)
- **Logical** (TRUE or FALSE)

The data type of a variable/constant in R can be identified using the built-in functions `class()` or `typeof()`. For example, the following variables and their values demonstrate different data types:

### 9.2.1 Numeric Data

In R, **numeric data** consists of two types:

- **Double (default)** – Represents floating-point numbers with decimal precision.
- **Integer** – Represents whole numbers, defined explicitly using the `L` suffix.

```
x <- 10 # Default: Double
y <- 10L # Explicit Integer
typeof(x) # "double"
```

```
[1] "double"
```

```
typeof(y) # "integer"
```

```
[1] "integer"
```

### 9.2.2 Character Data

R Strings can be created by assigning character values to a variable. These strings can be further concatenated by using various functions and methods to form a big string.

```
R program for String Creation

creating a string with double quotes
str1 <- "OK1"
cat ("String 1 is : ", str1)
```

String 1 is : OK1

```
creating a string with single quotes
str2 <- 'OK2'
cat ("String 2 is : ", str2)
```

String 2 is : OK2

```
str3 <- "This is 'acceptable and 'allowed' in R"
cat ("String 3 is : ", str3)
```

String 3 is : This is 'acceptable and 'allowed' in R

```
str4 <- 'Hi, Wondering "if this "works"'
cat ("String 4 is : ", str4)
```

String 4 is : Hi, Wondering "if this "works"

R Strings are always stored as double-quoted values. A double-quoted string can contain single quotes within it. Single-quoted strings can't contain single quotes. Similarly, double quotes can't be surrounded by double quotes.

### 9.2.3 Logical Data

In R, **logical data** consists of values that represent **Boolean (TRUE/FALSE) conditions**. Logical values are used in conditional statements, indexing, and logical operations.

R has **three logical values**: - TRUE (can be written as T) - FALSE (can be written as F) - NA (Logical missing value)

```
x <- TRUE
y <- FALSE
z <- NA # Missing logical value
```

```
typeof(x)
```

```
[1] "logical"
```

```
typeof(y)
```

```
[1] "logical"
```

```
typeof(z)
```

```
[1] "logical"
```

## 9.3 Variables

We have the following rules for a R variable name:

- A variable name must start with a letter and can be a combination of letters, digits, period(.) and underscore(\_). If it starts with period(.), it cannot be followed by a digit.
- A variable name cannot start with a number or underscore (\_)
- Variable names are case-sensitive (age, Age and AGE are three different variables)
- Reserved words cannot be used as variables (TRUE, FALSE, NULL, if...)

### 9.3.1 The assignment operator

#### 9.3.1.1 Using <- (Preferred Operator)

The <- operator is the standard way to assign values in R:

```
x <- 10
y <- "Hello, R!"
z <- TRUE
```

#### 9.3.1.2 Using = (Not Recommended)

Although = can be used for assignment, it is generally not recommended because it can cause issues in function arguments:

```
x = 10 # Works, but `<-` is preferred
```

**Best Practice:**

- Always use `<-` for assignments to avoid ambiguity.

## 9.4 Converting datatypes

Sometimes a value may have a datatype that is not suitable for using it. For example, consider the variable called `annual_income` in the code below:

```
annual_income <- "80000"
```

Suppose we wish to divide `annual_income` by 12 to get the monthly income. We cannot use the variable `annual_income` directly as its datatype is a string and not a number. Thus, numerical operations cannot be performed on the variable `annual_income`.

We'll need to convert `annual_income` to an integer. For that we will use the R's in-built `as.integer()` function:

```
annual_income <- as.integer(annual_income)
monthly_income <- annual_income/12
print(paste0("monthly income = ", monthly_income))
```

```
[1] "monthly income = 6666.66666666667"
```

Similarly, datatypes can be converted from one type to another using in-built R functions as shown below:

```
#Converting integer to character
as.character(9)
```

```
[1] "9"
```

```
#Converting character to numeric
as.numeric('9.4')
```

```
[1] 9.4
```

```
#Converting logical to integer
as.numeric(FALSE)
```

```
[1] 0
```

Note that any non-zero numeric value, if converted to the 'logical' datatype, will return **TRUE**, while converting 0 to the 'logical' datatype will return **FALSE**. Only numeric values can be converted to the 'logical' datatype.

```
Converting integer to logical
as.logical(40)
```

```
[1] TRUE
```

```
Converting integer to logical
as.logical(0)
```

```
[1] FALSE
```

```
Converting integer to logical
as.logical(-30.1)
```

```
[1] TRUE
```

Sometimes, conversion of a value may not be possible. For example, it is not possible to convert the variable `greeting` defined below to a number:

```
greeting <- "hello"
as.numeric(greeting)
```

```
Warning: NAs introduced by coercion
```

```
[1] NA
```

However, strings can be concatenated using the `paste0()` function:

```
paste0("hello", " there!")
```

```
[1] "hello there!"
```

The following table summarizes how to convert between **Numeric**, **Character**, and **Logical** types in R:

| From → To                         | Conversion Function          | Example Usage                                 | Notes                                   | Failure Behavior                             |
|-----------------------------------|------------------------------|-----------------------------------------------|-----------------------------------------|----------------------------------------------|
| <b>Numeric</b> → <b>Character</b> | <code>as.character(x)</code> | <code>as.character(42)</code><br>→ "42"       | Converts numbers to strings             | Not applicable (always succeeds)             |
| <b>Numeric</b> → <b>Logical</b>   | <code>as.logical(x)</code>   | <code>as.logical(0)</code><br>→ FALSE         | 0 is FALSE, non-zero is TRUE            | Returns NA if input is not numeric           |
| <b>Character</b> → <b>Numeric</b> | <code>as.numeric(x)</code>   | <code>as.numeric("3.14")</code><br>→ 3.14     | Returns NA if conversion fails          | Returns NA if conversion fails               |
| <b>Character</b> → <b>Logical</b> | <code>as.logical(x)</code>   | <code>as.logical("TRUE")</code><br>→ TRUE     | Case-sensitive, "TRUE" and "FALSE" work | Returns NA if input is not "TRUE" or "FALSE" |
| <b>Logical</b> → <b>Numeric</b>   | <code>as.numeric(x)</code>   | <code>as.numeric(TRUE)</code><br>→ 1          | TRUE = 1, FALSE = 0                     | Returns NA if input is not logical           |
| <b>Logical</b> → <b>Character</b> | <code>as.character(x)</code> | <code>as.character(FALSE)</code><br>→ "FALSE" | Converts logical values to strings      | Not applicable (always succeeds)             |

**Note:** Always verify conversions using `class()` or `typeof()` to ensure expected results.

## 9.5 typeof() vs. class() in R

In R, `typeof()` and `class()` are used to determine **different aspects of an object's type**:

- **typeof()**: Returns the **low-level storage mode** of an object.
- **class()**: Returns the **high-level classification** of an object.

```
x <- 10L # Integer value
typeof(x) # "integer"
```

```
[1] "integer"
```



```
class(x) # "integer"
```

```
[1] "integer"
```

```
y <- c(1, 2, 3) # Numeric vector
typeof(y) # "double"
```

```
[1] "double"
```

```
class(y) # "numeric"
```

```
[1] "numeric"
```

### 9.5.1 Key Differences:

| Function | Description                                       | Example Output                    |
|----------|---------------------------------------------------|-----------------------------------|
| typeof() | Shows how data is stored in memory                | "double", "integer", "character"  |
| class()  | Shows the high-level classification (for objects) | "numeric", "factor", "data.frame" |

### 9.5.2 Common Data Types

| Data Type         | typeof() Output | class() Output | Example                                                    |
|-------------------|-----------------|----------------|------------------------------------------------------------|
| <b>Numeric</b>    | "double"        | "numeric"      | x <- 3.14                                                  |
| <b>Integer</b>    | "integer"       | "integer"      | x <- 10L                                                   |
| <b>Character</b>  | "character"     | "character"    | x <-<br>"hello"                                            |
| <b>Logical</b>    | "logical"       | "logical"      | x <- TRUE                                                  |
| <b>Data Frame</b> | "list"          | "data.frame"   | x <-<br>data.frame(a<br>= 1:3, b =<br>c("A", "B",<br>"C")) |

## 9.6 Displaying information

### 9.6.1 Using `print()`

The `print()` function is the most basic way to display output.

```
x <- "Hello, R!"
print(x)
```

```
[1] "Hello, R!"
```

### 9.6.2 Using `cat()`

The `cat()` function concatenates and prints text without quotes.

```
name <- "Alice"
cat("Hello,", name, "!\n")
```

```
Hello, Alice !
```

- `cat()` does not return a value; it just displays output.
- `\n` adds a new line.

### 9.6.3 Using `paste()`

The `paste()` function concatenates text elements into a single string.

```
name <- "Alice"
paste("Hello,", name, "!\n")
```

```
[1] "Hello, Alice !\n"
```

- `paste()` returns a character string.
- `paste0()` is a variant of `paste()` that does not add spaces between elements.
- To print the result without quotes, use `cat()`

### 9.6.4 Using `message()`

The `message()` function is useful for warnings or informational messages.

```
message("This is a message!")
```

This is a message!

Unlike `print()`, `message()` does not print inside RMarkdown unless `message=TRUE` in chunk options.

### 9.6.5 Using `sprintf()`

For formatted output, use `sprintf()`:

```
name <- "Alice"
age <- 25
sprintf("My name is %s, and I am %d years old.", name, age)
```

```
[1] "My name is Alice, and I am 25 years old."
```

## 9.7 Taking user input

R's in-built `readline()` function can be used to accept an input from the user. For example, suppose we wish the user to input their age:

```
user_name <- readline(prompt="Enter your name: ")
```

Enter your name:

```
cat("Hello,", user_name, "!\n")
```

Hello, !

Since RMarkdown is non-interactive, `readline()` will not work inside a notebook. Instead, you can assign input directly for demonstration:

```
user_name <- "Alice"
cat("Hello,", user_name, "!\n")
```

Hello, Alice !

When using `readline()`, the input is always a *\*character string\**, and it must be converted explicitly to numeric before performing calculations.

```
age <- as.numeric(readline(prompt="Enter your age: "))
```

Enter your age:

```
cat("You are", age, "years old.\n")
```

You are NA years old.

## 9.8 Arithmetic Operations

R supports standard arithmetic operations for numeric values.

| Operation          | Symbol  | Example     | Result |
|--------------------|---------|-------------|--------|
| Addition           | +       | 5 + 3       | 8      |
| Subtraction        | -       | 10 - 4      | 6      |
| Multiplication     | *       | 6 * 2       | 12     |
| Division           | /       | 8 / 2       | 4      |
| Exponentiation     | ^ or ** | 3^2 or 3**2 | 9      |
| Integer Division   | %%/%    | 10 %/% 3    | 3      |
| Modulo (Remainder) | %%      | 10 %% 3     | 1      |

### Note:

- Integer division `%%/%` returns the quotient without the remainder.
- Modulo `%%` returns the remainder after division.

---

## 9.9 Comparison Operators

Comparison operations return `TRUE` or `FALSE`, often used for conditions.

| Operation                | Symbol | Example | Result |
|--------------------------|--------|---------|--------|
| Greater than             | >      | 5 > 3   | TRUE   |
| Less than                | <      | 2 < 1   | FALSE  |
| Greater than or equal to | >=     | 4 >= 4  | TRUE   |
| Less than or equal to    | <=     | 6 <= 5  | FALSE  |
| Equal to                 | ==     | 5 == 5  | TRUE   |
| Not equal to             | !=     | 3 != 2  | TRUE   |

**Note:**

- Always use == for comparison (not =).
- != checks if values are different.

## 9.10 Logical Operators

Logical operators are used to combine conditions in R. There are two types of logical operators:

Element-wise operators: & (AND), | (OR), and ! (NOT) – work element-by-element for vectors. Short-circuit operators: && (AND), || (OR) – only evaluate the first element of each condition, primarily used in control flow (e.g., if statements).

| Operator                 | Symbol | Description                                 | Example           | Result |
|--------------------------|--------|---------------------------------------------|-------------------|--------|
| <b>AND</b>               | &      | Element-wise AND (Both must be TRUE)        | (5 > 3) & (2 < 4) | TRUE   |
| <b>OR</b>                |        | Element-wise OR (At least one must be TRUE) | (5 > 3)   (2 > 4) | TRUE   |
| <b>NOT</b>               | !      | Negates a logical value                     | !(5 > 3)          | FALSE  |
| <b>Short-circuit AND</b> | &&     | Evaluates only the first element            | TRUE && FALSE     | FALSE  |
| <b>Short-circuit OR</b>  |        | Evaluates only the first element            | TRUE    FALSE     | TRUE   |

\*\*Difference between element-wise and short-circuit operators

## 9.11 2. Differences Between Element-wise and Short-circuit Operators

| Operator Type     | Symbol | Works on Vectors?       | Use Case                        |
|-------------------|--------|-------------------------|---------------------------------|
| Element-wise AND  | &      | Yes                     | Use with vectors or data frames |
| Element-wise OR   |        | Yes                     | Use with vectors or data frames |
| Short-circuit AND | &&     | No (only first element) | Use in if statements            |
| Short-circuit OR  |        | No (only first element) | Use in if statements            |

### Key Takeaways:

- Use & and | for **vector operations** (e.g., filtering in data frames).
- Use && and || in if statements for **better efficiency**. - The ! operator **negates logical values** (useful for filtering and reversing conditions).

Examples:

```
x <- c(TRUE, FALSE, TRUE)
y <- c(FALSE, TRUE, TRUE)

Element-wise AND
x & y
```

```
[1] FALSE FALSE TRUE
```

```
Element-wise OR
x | y
```

```
[1] TRUE TRUE TRUE
```

```
a <- 10
b <- 5

if (a > 0 && b > 0) {
 print("Both are positive")
}
```

```
[1] "Both are positive"
```

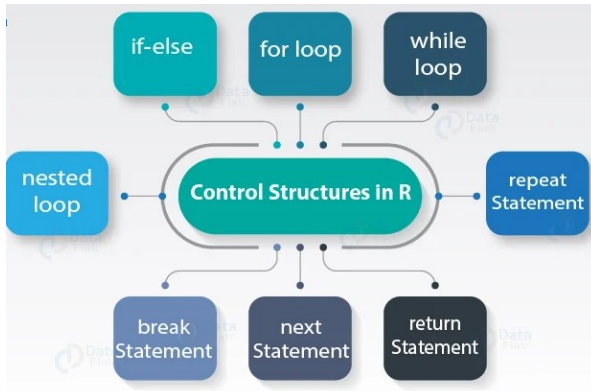
```
if (a > 0 || b < 0) {
 print("At least one condition is met")
}
```

```
[1] "At least one condition is met"
```

```
x <- c(TRUE, FALSE, TRUE)
!x # [FALSE TRUE FALSE]
```

```
[1] FALSE TRUE FALSE
```

## 10 R: Control flow tools



### 10.1 Code Blocks: R vs. Python

Different programming languages define code blocks differently:

- **Python** uses **indentation** to structure code.
- **R** uses **curly braces {}** to define code blocks.

You have learned **indentation** (spaces or tabs) determines the structure of code in Python.

#### 10.1.1 Code blocks in R (Uses {})

In R, **curly braces {}** define code blocks, and indentation is not required (but recommended for readability).

```
for (i in 1:5) {
 print(i)
}
```

```
[1] 1
[1] 2
[1] 3
```



```
[1] 4
[1] 5
```

Another key difference in control flow syntax between Python and R is how conditions are written. In Python, conditions do not require parentheses, whereas in R, they must be enclosed in parentheses.

## 10.2 TryCatch

When running R code, errors can occur due to unexpected inputs, missing data, or issues with computations. Without proper error handling, an entire execution may be stopped due to an error. The `tryCatch` function in R provides a mechanism to handle errors gracefully, allowing execution to continue and custom responses to be provided.

### Basic Syntax

```
tryCatch({
 # Code that may generate an error
}, error = function(e) {
 # Code to handle errors
}, warning = function(w) {
 # Code to handle warnings
}, finally = {
 # Code that will always execute
})
```

NULL

- `error`: Handles errors that stop execution.
- `warning`: Handles warnings that don't necessarily stop execution.
- `finally`: Executes code regardless of whether an error occurs.

**Example:** Input an integer from the user. If the user inputs a valid integer, print whether it is a multiple of 3. However, if the user does not input a valid integer, print a message saying that the input is invalid.

```
num <- "3r"
tryCatch({
 num_int <- as.integer(num)
 if (num_int %% 3 == 0) {
 print("Number is a multiple of 3")
 }
})
```

```

} else {
 print("Number is not a multiple of 3")
}
}, error = function(e) {
 print("Error encountered")
}, warning = function(w) {
 print("Warning encountered")
})

```

```
[1] "Warning encountered"
```

The warning block will be triggered, not the error block.

### Why

- `as.integer("3r")` does not generate an error. Instead, it produces a warning and returns NA.
- In R, an error occurs when an operation stops execution (e.g., division by zero in integer operations).
- A warning occurs when something unexpected happens, but execution continues (e.g., invalid conversion).

## 10.3 Control flow tools

Control flow in R relies on comparison operators and logical operators to make decisions based on conditions. These operators are commonly used in conditional statements like `if`, `ifelse()`, and loops.

- **Comparison operators** are used to compare values and return a logical value (`TRUE` or `FALSE`).
- **Logical operators** are used to combine multiple conditions in control flow statements.

Like Python, R provides `if-else`, `for` loops, `while` loops, and `switch/case` for decision-making and iteration. However, in Python, conditions do not require parentheses, whereas in R, parentheses must enclose the condition.

```

x <- 5
while (x > 0) { # Parentheses required
 print(x)
 x <- x - 1
}

```

```
[1] 5
[1] 4
[1] 3
[1] 2
[1] 1
```

## 10.4 Conditional statements

The `if-else if-else` statements can check several conditions, and execute the code corresponding to the condition that is true. Note that there can be as many `else if` statements as required.

**Example:** Assign letter grade based on a student's final score

```
score <- 75 # Change this value to test different cases

if (score >= 90) {
 print("Grade: A")
} else if (score >= 80) {
 print("Grade: B")
} else if (score >= 70) {
 print("Grade: C")
} else if (score >= 60) {
 print("Grade: D")
} else {
 print("Grade: F")
}
```

```
[1] "Grade: C"
```

## 10.5 Loops

### 10.5.1 The `:` operator

The `:` operator in R is used to generate sequences of integers. It is similar to Python's `range()` function, but with some key differences.

#### Basic Syntax

`start:end`

- Generates a sequence of integers from `start` to `end`.
- Inclusive: Both `start` and `end` are included.

### Examples

```
1:5
```

```
[1] 1 2 3 4 5
```

In Python, the equivalent would be `list(range(1, 6))`

#### 10.5.1.1 Reverse sequence

The `:` operator also works in Reverse

```
Counting down
5:1
```

```
[1] 5 4 3 2 1
```

#### 10.5.1.2 Using `:` with negative numbers

```
Counting down
-3:3
```

```
[1] -3 -2 -1 0 1 2 3
```

#### 10.5.1.3 Using `:` with Non-integers

If the start or end value is not an integer, R coerces it to an integer.

```
The decimal values are truncated to integers before generating the sequence
1.5:5.5
```

```
[1] 1.5 2.5 3.5 4.5 5.5
```

## 10.5.2 seq() function

Difference: ``: only creates sequences with a step size of 1, while `seq()` allows custom increments.

```
Counting by 2
seq(1, 10, by = 2)
```

```
[1] 1 3 5 7 9
```

### Summary

| Feature                      | R (: Operator)      | Python (range())      | R (seq() Function)         |
|------------------------------|---------------------|-----------------------|----------------------------|
| <b>Generates integers</b>    | Yes                 | Yes                   | Yes                        |
| <b>Inclusive of end?</b>     | Yes                 | No (excludes end)     | Yes                        |
| <b>Works in reverse?</b>     | Yes (5:1)           | Yes (range(5, 0, -1)) | Yes (seq(5, 1, by = -1))   |
| <b>Handles non-integers?</b> | Converts to integer | Must be integer       | Supports non-integer steps |
| <b>Custom step size?</b>     | No (always 1)       | Yes (range(1, 10, 2)) | Yes (seq(1, 10, by = 2))   |

## 10.5.3 for loop

The `for` loop is used when you need to iterate over a sequence of numbers, vectors, lists, or other iterable objects.

### 10.5.3.1 Using : in for loops

Since `:` creates integer sequences, it is commonly used in `for` loops.

```
for (i in 1:5) {
 print(i)
}
```

```
[1] 1
[1] 2
[1] 3
[1] 4
[1] 5
```

**Example:** Print the first  $n$  elements of the Fibonacci sequence, where  $n$  is an integer input by the user, such that  $n > 2$ . In a fibonacci sequence, each number is the sum of the preceding two numbers, and the sequence starts from 0,1. The sequence is as follows:

0, 1, 1, 2, 3, 5, 8, 13, ....

```
n <- readline("Enter a number:")
```

Enter a number:

```
n <- 10
```

```
Define the number of terms
if (n <= 0) {
 stop("n must be a positive integer")
}

Initialize the sequence with the first two Fibonacci numbers
n1 <- 0
n2 <- 1
elements <- c(n1, n2)

Generate the Fibonacci sequence iteratively
for (i in 3:n) {
 n3 <- n1 + n2 # Compute the next term
 elements <- c(elements, n3) # Append to the sequence

 # Shift values for the next iteration
 n1 <- n2
 n2 <- n3
}

Print the Fibonacci sequence
print(elements)
```

```
[1] 0 1 1 2 3 5 8 13 21 34
```

```
cat("These are the first", n, "elements of the Fibonacci series.\n")
```

These are the first 10 elements of the Fibonacci series.

- The `stop()` function in R is used to terminate execution and display an error message when a certain condition is met. It is often used for error handling to prevent further execution of the script when an invalid input is detected.
- The `c()` function in R is used to combine values into a vector. It is one of the most fundamental functions in R, as vectors are the primary data structure. You can think of vectors as being similar to Python lists.

### 10.5.4 while loop

The `while` loop executes as long as the specified condition evaluates to `TRUE`.

**Example:** Print all the elements of the Fibonacci sequence less than `n`, where `n` is an integer input by the user, such that `n > 2`. In a fibonacci sequence, each number is the sum of the preceding two numbers, and the sequence starts from 0,1. The sequence is as follows:

0,1,1,2,3,5,8,13,..

```
n = readline("Enter a number:")
```

Enter a number:

```
n = 50
```

```
Initialize the sequence with the first number
n1 <- 0
n2 <- 1

Store the Fibonacci sequence
elements <- n1

Generate Fibonacci numbers less than n
while (n2 < n) {
 elements <- c(elements, n2) # Append the next number

 # Compute the next Fibonacci number
 n3 <- n1 + n2
```

```

Shift values for the next iteration
n1 <- n2
n2 <- n3
}

Print the Fibonacci sequence
print(elements)

```

```
[1] 0 1 1 2 3 5 8 13 21 34
```

```
cat("These are all the Fibonacci numbers less than", n, "\n")
```

These are all the Fibonacci numbers less than 50

## 10.6 Loop control statements

in R, there are two statements that are commonly used to control the execution of loops by altering their normal flow

### 10.6.1 break statement (Loop Termination)

The **break** statement is used to immediately exits the loop when certain condition is met. It will stop execution of the loop entirely

For example, suppose we need to keep asking the user to input year of birth and compute the corresponding age, until the user enters 1900 as the year of birth.

```

Get current year dynamically
current_year <- as.integer(format(Sys.Date(), "%Y"))

Check if running in interactive mode
if (interactive()) {
 while (TRUE) {
 user_input <- readline("Enter year of birth (1900-present) or 'q' to quit: ")

 if (tolower(user_input) == "q") {
 cat("Goodbye!\n")
 break
 }
 }
}

```



```

year <- suppressWarnings(as.integer(user_input))

if (is.na(year)) {
 cat("Error: Please enter a valid year as a number.\n")
} else if (year == 1900) {
 cat("Special year entered. Exiting program.\n")
 break
} else if (year < 1900) {
 cat("Error: Year must be 1900 or later.\n")
} else if (year > current_year) {
 cat("Error: Year cannot be in the future.\n")
} else {
 age <- current_year - year
 cat(sprintf("Based on birth year %d, age in %d = %d years\n",
 year, current_year, age))
}
}
} else {
 cat("Skipping user input section since Quarto is running in batch mode.\n")
}

```

### 10.6.2 next statement (Skip Iteration, Similar to continue in Python)

The `next` statement is used to skip the current iteration and moves to the next one. It does not exit the loop but skips the rest of the current iteration.

For example, consider the following code:

```

for (i in 1:10) {
 if (i == 5) {
 next # Skip iteration when i is 5
 }
 print(i)
}

```

```

[1] 1
[1] 2
[1] 3
[1] 4
[1] 6
[1] 7

```

```
[1] 8
[1] 9
[1] 10
```

When the control flow reads the statement `next`, it goes back to the beginning of the `for` loop, and ignores the lines of code below the statement.

### 10.6.3 Practice exercise

Write a program that asks the user the question, “How many stars are in the Milky Way (in billions)?”. If the user answers 100, the program should print correct, and stop. However, if the user answers incorrectly, the program should print “incorrect”, and ask them if they want to try again. The program should continue to run until the user answers correctly, or they want to stop trying.

```
while (TRUE) {
 answer = readline("How many stars are there in the Milky Way (in billions)? ")
 if (answer == '100') {
 print("Correct")
 break
 } else {
 print("Incorrect")
 }
 try_again = readline("Do you want to try again? (Y/N):")
 if (try_again == 'Y') {
 next
 } else {
 break
 }
}
```

## 10.7 characters in R

### 10.7.1 Basic Definitions

- In **Python**, strings are defined using either single (') or double (") quotes.
- In **R**, character values (strings) are always enclosed in double quotes (" "), though single quotes (' ') also work, But single quotes are normally only used to delimit character constants containing double quotes.

### 10.7.2 String/Character Length

- In Python, `len()` is used to count the number of characters.
- In R, `nchar()` is used.

```
s <- "Hello"
print(nchar(s))
```

```
[1] 5
```

### 10.7.3 String Indexing and Slicing

- Python allows direct indexing and slicing using `[]`.
- R does not support direct indexing for characters. Instead, `substr()` is used.

```
s <- "Hello"
print(substr(s, 2, 2)) # Output: "e" (1-based index)
```

```
[1] "e"
```

```
print(substr(s, 2, 4))
```

```
[1] "ell"
```

**Key Difference:** Python uses 0-based indexing, while R uses 1-based indexing.

### 10.7.4 String Concatenation

- Python: Uses `+` for concatenation.
- R: Uses `paste()` or `paste0()`.

```
s1 <- "Hello"
s2 <- "World"
print(paste(s1, s2))
```

```
[1] "Hello World"
```

```
print(paste0(s1, s2))
```

```
[1] "HelloWorld"
```

### 10.7.5 Changing Case (Uppercase & Lowercase)

- Python: `s.upper()` and `s.lower()` for conversion
- R: `toupper(s)` and `tolower(s)`

```
s <- "Hello"
print(toupper(s))
```

```
[1] "HELLO"
```

```
print(tolower(s))
```

```
[1] "hello"
```

### 10.7.6 Finding Substrings

- Python: Uses `.find()` or `in` to check for substrings.
- R: Uses `grep()` or `grepl()`.

```
s <- "Hello World"
print(grepl("World", s))
```

```
[1] TRUE
```

```
print(grep("World", s))
```

```
[1] 1
```

### 10.7.7 String Replacement

- Python: Uses `.replace()`.
- R: Uses `gsub()`.

### 10.7.8 String Splitting

- Python: Uses `split()` method in strings to break them into lists
- R: Uses `strsplit()` function to split strings into a list of character vectors.

```
sentence <- "Hello, how are you?"
words <- strsplit(sentence, " ") # Split by space
print(words)
```

```
[[1]]
[1] "Hello," "how" "are" "you?"
```

`strsplit()` returns a list, even if applied to a single string. The delimiter is treated as a regex, meaning special characters need to be escaped

```
splitting using a comma
sentence <- "apple,banana,grape,orange"
words <- strsplit(sentence, ",")
print(words[[1]])
```

```
[1] "apple" "banana" "grape" "orange"
```

### 10.7.9 Summary table

| Feature                   | Python (String)                                          | R (Character)                                                     |
|---------------------------|----------------------------------------------------------|-------------------------------------------------------------------|
| <b>Definition</b>         | 'Hello' or "Hello"                                       | "Hello"                                                           |
| <b>Data Type Check</b>    | <code>type("Hello") → str</code>                         | <code>class("Hello") → "character"</code>                         |
| <b>Length</b>             | <code>len(s)</code>                                      | <code>nchar(s)</code>                                             |
| <b>Indexing</b>           | <code>s[1]</code> (0-based)                              | <code>substr(s, 2, 2)</code> (1-based)                            |
| <b>Concatenation</b>      | <code>s1 + s2</code>                                     | <code>paste(s1, s2)</code> or <code>paste0(s1, s2)</code>         |
| <b>Upper/Lower Case</b>   | <code>s.upper()</code> , <code>s.lower()</code>          | <code>toupper(s)</code> , <code>tolower(s)</code>                 |
| <b>Finding Substrings</b> | "World" in <code>s</code> , <code>s.find("World")</code> | <code>grepl("World", s)</code> ,<br><code>grep("World", s)</code> |
| <b>Replace Substring</b>  | <code>s.replace("old", "new")</code>                     | <code>gsub("old", "new", s)</code>                                |
| <b>Splitting</b>          | <code>s.split(" ")</code> (default: whitespace)          | <code>strsplit(s, " ")</code> (default: whitespace)               |

## 10.8 Practice exercises

### 10.8.1 Practice exercise 1

Try converting the following Python string operations into equivalent R code:

```
s = "Data Science"
print(len(s))
print(s[5:])
print(s.replace("Data", "AI"))
```

### 10.8.2 Practice exercise 2

Define a character and count the number of *ts*.

```
char_vec <- 'Getting a tatto is not a nice experience'
#Initializing a variable 'count_t' which will store the number of 't's in the string
count_t <- 0

#Iterating over the entire length of the string.
#The length of the string is given by the len() function
for (i in 1:nchar(char_vec)) {
 if (substr(char_vec, i, i) == 't') {
 count_t <- count_t + 1
 }
}
print(paste("Number of 't's in the string = ", count_t))
```

```
[1] "Number of 't's in the string = 6"
```

Use `strsplit()` to implement this function.

### 10.8.3 Practice exercise 3

Write a program that prints the number of 'the's found in sentence

```

sentence <- "She sells the sea shells on the sea shore during the summer"
count <- 0
for (i in 1:(nchar(sentence) - nchar("the"))) {
 if (substr(sentence, i, i + nchar("the") - 1) == "the") {
 count <- count + 1
 }
}
print(paste("Number of thes in the sentence = ", count))

```

```
[1] "Number of thes in the sentence = 3"
```

Use `strsplit()` to implement this function

## 10.8.4 Practice exercise 4

### 10.8.4.1 Problem 1

```
my_num <- "0"
```

Using `my_num` above:

- check if the number is numeric
- convert the number to numeric

### 10.8.4.2 Problem 2

- Define a variable to be your name.
- Define a variable to be your favorite number.
- Define a variable to be your favorite color.
- Print a sentence that reads: “My name is {} and my favorite number is {}”.
- Store a variable that checks if your favorite number is less than 10. Print this variable.
- Store a variable that checks if your favorite number is even. Print this variable.
- Store a variable that checks if your favorite color is “red”. Print this variable.
- Convert one of your stored Boolean to a numeric data type. What happens?

- Print a statement that returns TRUE if your favorite number is less than 10 and favorite color is NOT red and returns FALSE otherwise.
- Print a statement that return TRUE if your favorite number is even or less than 10.

### 10.8.5 Practice exercise 5

Create an interactive R code block that asks the user to enter their age and ensures the input is a **valid integer**. The code should handle incorrect inputs and guide the user until they provide a valid response.

#### Requirements

- The script should **continuously prompt the user** until they enter a valid integer.
- If the input is **not an integer**, the script should display an error message and ask again.
- Once a valid integer is entered:
  - If the age is **18 or older**, display: "You are an adult."
  - If the age is **less than 18**, display: "You are a minor."

#### Guidelines & Hints

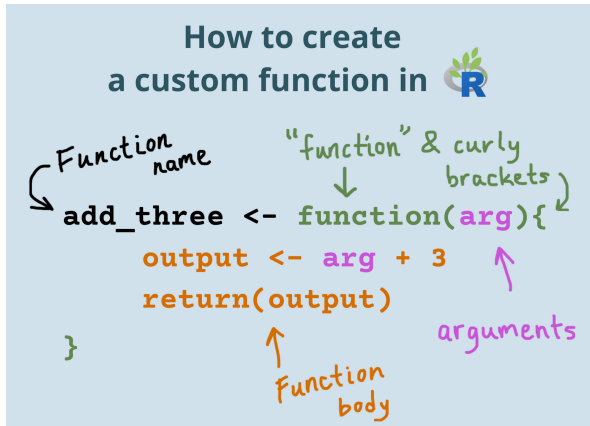
- Use **readline()** to collect user input.
- Convert the input using **as.integer()** and check if it returns **NA** for invalid inputs.
- Utilize a **loop** to ensure the user is repeatedly prompted until they enter a valid integer.
- Provide **clear error messages** to guide the user.

#### Example User Interaction

| Input    | Output                                  |
|----------|-----------------------------------------|
| "abc"    | "Please enter a valid integer for age." |
| "twenty" | "Please enter a valid integer for age." |
| "18"     | "You are an adult."                     |
| "17"     | "You are a minor."                      |



# 11 R: Functions



## 11.1 Functions in R

Functions in R are used to **encapsulate reusable code**. R provides a wide range of **built-in functions** for mathematical operations, statistics, and data manipulation, and users can also define their own functions.

---

### 11.1.1 Built-in Functions in R

R is a language specifically designed for statistical computing, with many of its core libraries developed by leading statisticians. It excels in statistical analysis and is widely used in academic research and applied statistics. While Python is more versatile for general-purpose programming and machine learning, R remains a strong choice for pure statistical work.

R has many **predefined functions** for mathematical operations, statistics, and data manipulation. These functions are part of R's base and default packages, allowing users to perform various tasks without additional libraries.

### 11.1.1.1 mathematical Functions

| Function                    | Description                                            |
|-----------------------------|--------------------------------------------------------|
| <code>abs(x)</code>         | Absolute value of <code>x</code>                       |
| <code>sqrt(x)</code>        | Square root of <code>x</code>                          |
| <code>exp(x)</code>         | Exponential function ( $e^x$ )                         |
| <code>log(x, base=n)</code> | Logarithm (default natural log)                        |
| <code>log10(x)</code>       | Base-10 logarithm                                      |
| <code>log2(x)</code>        | Base-2 logarithm                                       |
| <code>round(x, n)</code>    | Rounds <code>x</code> to <code>n</code> decimal places |
| <code>floor(x)</code>       | Rounds <code>x</code> down                             |
| <code>ceiling(x)</code>     | Rounds <code>x</code> up                               |
| <code>trunc(x)</code>       | Truncates <code>x</code> (removes decimal part)        |

```
sqrt(25)
```

```
[1] 5
```

```
abs(-10)
```

```
[1] 10
```

```
round(3.14159, 2)
```

```
[1] 3.14
```

### 11.1.1.2 Statistical Functions

| Function               | Description                               |
|------------------------|-------------------------------------------|
| <code>mean(x)</code>   | Mean of <code>x</code>                    |
| <code>median(x)</code> | Median of <code>x</code>                  |
| <code>var(x)</code>    | Variance of <code>x</code>                |
| <code>sd(x)</code>     | Standard deviation of <code>x</code>      |
| <code>sum(x)</code>    | Sum of all elements in <code>x</code>     |
| <code>prod(x)</code>   | Product of all elements in <code>x</code> |
| <code>min(x)</code>    | Minimum value in <code>x</code>           |
| <code>max(x)</code>    | Maximum value in <code>x</code>           |

| Function                    | Description                 |
|-----------------------------|-----------------------------|
| <code>range(x)</code>       | Minimum and maximum of x    |
| <code>quantile(x, p)</code> | p-th quantile of x          |
| <code>cor(x, y)</code>      | Correlation between x and y |
| <code>cov(x, y)</code>      | Covariance between x and y  |

```
Create a sample numeric vector
x <- c(10, 20, 30, 40, 50, 60, 70, 80, 90, 100)
y <- c(15, 25, 35, 45, 55, 65, 75, 85, 95, 105)

1. Basic statistical functions
mean_x <- mean(x) # Mean of x
median_x <- median(x) # Median of x
var_x <- var(x) # Variance of x
sd_x <- sd(x) # Standard deviation of x

2. Summation and product
sum_x <- sum(x) # Sum of elements in x
prod_x <- prod(x[1:5]) # Product of first five elements of x

3. Minimum, Maximum, and Range
min_x <- min(x) # Minimum value
max_x <- max(x) # Maximum value
range_x <- range(x) # Range (min and max)

4. Quantiles
quantiles_x <- quantile(x, probs = c(0.25, 0.5, 0.75)) # 25th, 50th, 75th percentile

5. Correlation and Covariance
cor_xy <- cor(x, y) # Correlation between x and y
cov_xy <- cov(x, y) # Covariance between x and y

Print results
cat("Mean of x:", mean_x, "\n")
```

Mean of x: 55

```
cat("Median of x:", median_x, "\n")
```

Median of x: 55

```
cat("Variance of x:", var_x, "\n")
```

Variance of x: 916.6667

```
cat("Standard deviation of x:", sd_x, "\n\n")
```

Standard deviation of x: 30.2765

```
cat("Sum of x:", sum_x, "\n")
```

Sum of x: 550

```
cat("Product of first five elements in x:", prod_x, "\n\n")
```

Product of first five elements in x: 1.2e+07

```
cat("Min of x:", min_x, "\n")
```

Min of x: 10

```
cat("Max of x:", max_x, "\n")
```

Max of x: 100

```
cat("Range of x:", range_x, "\n\n")
```

Range of x: 10 100

```
cat("Quantiles of x:\n")
```

Quantiles of x:

```
print(quantiles_x)
```

```
 25% 50% 75%
32.5 55.0 77.5
```

```
cat("\nCorrelation between x and y:", cor_xy, "\n")
```

Correlation between x and y: 1

```
cat("Covariance between x and y:", cov_xy, "\n")
```

Covariance between x and y: 916.6667

### 11.1.1.3 Random number generation

R provides various functions to generate random numbers from different distributions. These functions are essential for **simulation, statistical modeling, and machine learning**.

| Function                              | Description           | Example                                    |
|---------------------------------------|-----------------------|--------------------------------------------|
| <code>runif(n, min, max)</code>       | Uniform distribution  | <code>runif(5, 0, 10)</code>               |
| <code>rnorm(n, mean, sd)</code>       | Normal distribution   | <code>rnorm(5, mean=0, sd=1)</code>        |
| <code>rbinom(n, size, prob)</code>    | Binomial distribution | <code>rbinom(5, size=10, prob=0.5)</code>  |
| <code>rpois(n, lambda)</code>         | Poisson distribution  | <code>rpois(5, lambda=3)</code>            |
| <code>sample(x, size, replace)</code> | Random sampling       | <code>sample(1:10, 5, replace=TRUE)</code> |

#### 11.1.1.3.1 Setting a seed for reproducibility

Setting a seed ensures that random number generation produces the same results every time.

```
set.seed(42)
runif(3)
```

```
[1] 0.9148060 0.9370754 0.2861395
```

#### 11.1.1.3.2 Generating uniform random numbers (`runif()`)

The `runif()` function generates random numbers from a **uniform distribution** between a given `min` and `max`.

```
set.seed(42) # Set seed for reproducibility
runif(5, min=0, max=10)
```

```
[1] 9.148060 9.370754 2.861395 8.304476 6.417455
```

#### 11.1.1.3.3 Generating Normally Distributed Numbers (`rnorm()`)

The `rnorm()` function generates random numbers from a normal (Gaussian) distribution.

```
set.seed(42)
rnorm(5, mean=0, sd=1)
```

```
[1] 1.3709584 -0.5646982 0.3631284 0.6328626 0.4042683
```

#### 11.1.1.3.4 Random Sampling (`sample()`)

The `sample()` function randomly selects elements from a given vector.

```
set.seed(42)
sample(1:10, 5, replace=TRUE)
```

```
[1] 1 5 1 9 10
```

## 11.2 User-defined Functions

### 11.2.1 Defining a function

Functions in R are defined using the keyword `function()`. All the statements within a function are enclosed with `{}` braces.

```
my_function <- function() { # create a function with the name my_function
 print("Hello R!")
}
```

### 11.2.2 Calling a function

After creating a Function, you have to call the function to use it.

```
my_function() # call the function named my_function
```

```
[1] "Hello R!"
```

### 11.2.3 Arguments

Information can be passed into functions as arguments.

Arguments are specified after the function name, inside the parentheses. You can add as many arguments as you want, just separate them with a comma.

The following example has a function with one argument (fname). When the function is called, we pass along a first name, which is used inside the function to print the full name:

```
my_function <- function(fname) {
 paste("Hello", fname)
}
```

```
my_function("Lizhen")
```

```
[1] "Hello Lizhen"
```

```
my_function("Sarah")
```

```
[1] "Hello Sarah"
```

```
my_function("Jack")
```

```
[1] "Hello Jack"
```

### 11.2.4 Return values

To let a function return a result, use the `return()` function:

```
Function to return the largest of three numbers
largest_value <- function(a, b, c) {
 return(max(a, b, c))
}

Example usage
result <- largest_value(10, 25, 15)
print(result)
```

```
[1] 25
```

A function in R does not necessarily need to return a value explicitly; it can simply print output to the console. Look at the function defined below. It takes an integer as an argument, and prints whether the integer is odd or even.

```
odd_even <- function(intgr) {
 if (intgr %% 2 == 0) {
 print("even")
 } else {
 print("odd")
 }
}

odd_even(3)
```

```
[1] "odd"
```

## 11.3 Function arguments

Like Python, functions in R support multiple types of arguments, including **positional arguments**, **default arguments**, **variable-length arguments**, and **keyword arguments**. The behavior of function arguments in R is nearly identical to Python.

### 11.3.1 Positional arguments

Write a function that returns all prime numbers between  $a$  and  $b$ , where  $a$  and  $b$  are parameters of the function.



```

prime <- function(a, b) {
 prime_numbers <- c()
 for (number in a:b) {
 prime = 1

 for (factor in 2:(number - 1)) {
 if (number %% factor == 0) {
 prime = 0
 }
 }

 if (prime == 1) prime_numbers <- c(prime_numbers, number)
 }
 return(prime_numbers)
}
prime(40, 60)

```

```
[1] 41 43 47 53 59
```

### 11.3.2 default arguments

```

greet <- function(name = "Guest", age = 25, country = "USA") {
 cat("Hello,", name, "!\n")
 cat("You are", age, "years old and from", country, ".\n")
}

Calling the function with default values
greet()

```

```

Hello, Guest !
You are 25 years old and from USA .

```

### 11.3.3 keyword arguments

```

greet <- function(name = "Guest", age = 25, country = "USA") {
 cat("Hello,", name, "!\n")
 cat("You are", age, "years old and from", country, ".\n")
}

```

```
}

Calling the function with default values
greet()
```

```
Hello, Guest !
You are 25 years old and from USA .
```

```
Calling the function with keyword arguments
greet(name = "Alice", age = 30, country = "Canada")
```

```
Hello, Alice !
You are 30 years old and from Canada .
```

```
Calling with positional arguments (order matters)
greet("Bob", 40, "UK")
```

```
Hello, Bob !
You are 40 years old and from UK .
```

```
Mixing positional and named arguments
greet("Charlie", country = "Australia")
```

```
Hello, Charlie !
You are 25 years old and from Australia .
```

### 11.3.4 Variable-number arguments

```
Function that accepts variable-length arguments and computes their sum
sum_values <- function(...) {
 values <- c(...) # Collect arguments into a vector
 sum(values, na.rm = TRUE) # Sum with NA removal support
}

Example usage
result <- sum_values(10, 20, 30, 40, 50)
print(result)
```

[1] 150

```
Function that accepts extra named arguments
display_info <- function(name, ...) {
 cat("Name:", name, "\n")

 # Capture extra arguments as a list
 extras <- list(...)
 for (key in names(extras)) {
 cat(key, ":", extras[[key]], "\n")
 }
}

Example usage with additional named arguments
display_info("David", age = 28, country = "Germany", occupation = "Engineer")
```

```
Name: David
age : 28
country : Germany
occupation : Engineer
```

## 11.4 Variable scope

R has global and local variables, similar to Python, but there are some differences in how they are managed.

### 11.4.1 Local variables

- Variables declared inside a function are local to that function.
- They are not accessible outside the function.

```
my_function <- function() {
 x <- 10 # Local variable
 cat("Inside function: x =", x, "\n")
}

my_function()
print(x) # Error: object 'x' not found
```

### 11.4.2 Global variables

- Variables declared outside any function are global.
- They can be accessed from anywhere unless shadowed by a local variable of the same name.

```
y <- 100 # Global variable

my_function <- function() {
 cat("Inside function: y =", y, "\n") # Can access global variable
}

my_function()
```

Inside function: y = 100

```
print(y) # Still accessible
```

[1] 100

### 11.4.3 Modifying Global variables inside a function

Unlike Python (`global` keyword), R requires the special assignment operator `<<-` to modify a global variable inside a function.

```
counter <- 0 # Global variable

increase_counter <- function() {
 counter <<- counter + 1 # Modifies global variable
}

increase_counter()
increase_counter()
print(counter)
```

[1] 2

## 11.5 Practice exercises

### 11.5.1 Problem 1

Write a function where the input is any word and the output is the number of letters in it. Store the output of the function, then print the sentence: “{} has {} letters”.

### 11.5.2 Problem 2

Write a function that has no input values and simulates rolling a die. In other words, it should generate a random integer between 1 and 6, inclusive. The function should return the integer. Then use a loop to “roll the die” 5 times.

### 11.5.3 Problem 3

Write a function that simulates rolling a die. The function should have an input for the number of sides of the die and the number of times the die is rolled. Have the default number of sides be 6. Within the function calculate the `mean()`, `sum()`, `min()`, `max()` of all the dice rolled.

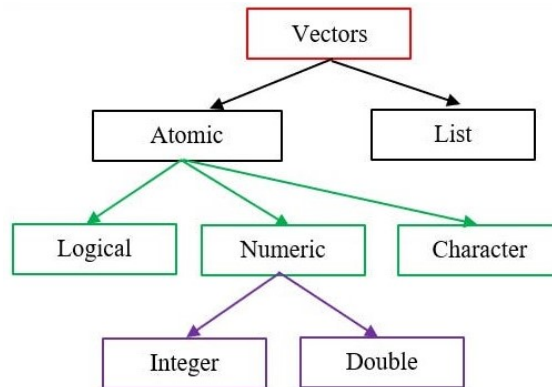
The function should return the dice rolled and the 4 calculations. Run your function with any sided dice and any number of times.

Below is my function and make it work

```
roll_die <- function(size=6, roll_times=1){
 return (sample(1:size, roll_times, replace=TRUE))
}

paste("The average of", roll_times, "rolling a die with", size, "is", mean(roll_die(roll_times)))
```

## 12 R: Atomic vectors



**Atomic vectors** are the most basic data type in R.

An atomic vector is a one-dimensional collection of elements, where all elements must have the same data type. If you assign different data type values to an atomic vector, the vector will coerce all elements to the most flexible common data type. This is known as type coercion in R.

### 12.0.1 Creating a vector

Atomic vectors are created using the `c()` function, which stands for **combine** or **concatenate**

```
numeric_vec <- c(1, 2.5, 3.8)
print(numeric_vec)
```

```
[1] 1.0 2.5 3.8
```

```
Character vector
char_vec <- c("apple", "banana", "cherry")
print(char_vec)
```

```
[1] "apple" "banana" "cherry"
```

### 12.0.1.1 Type Coercion (mixed types)

- When you assign elements of different types to an atomic vector, R does **not** raise an error.
- Instead, it **automatically performs type coercion** to ensure all elements in the vector share the same type.
- Coercion follows a hierarchy: **Logical** → **Integer** → **Double** → **Character**.

```
vec <- c(1, "apple", TRUE) # Mixed types
typeof(vec)
```

```
[1] "character"
```

### 12.0.2 Atomic vector types

- R has **six** atomic vector types:
  1. **Logical** (TRUE, FALSE)
  2. **Integer** (Whole numbers: 1L, 2L, -5L)
  3. **Double (Numeric)** (Decimal numbers: 1.5, 3.14, -2.8)
  4. **Character** (Text strings: "hello", "R programming")
  5. **Complex** (Complex numbers: 1 + 2i)
  6. **Raw** (Stores raw bytes)
- In this course, we normally use the first four types: **Logical**, **Integer**, **Double**, and **Character**.

#### 12.0.2.1 Checking Whether a Variable is an Atomic Vector

There are two common functions to check if a variable is an atomic vector:

- `is.atomic(x)`: Returns TRUE if `x` is an atomic object.
- `is.vector(x)`: Returns TRUE if `x` is a vector (with no attributes other than names).

```
is.atomic(numeric_vec)
```

```
[1] TRUE
```

```
is.vector(char_vec)
```

```
[1] TRUE
```

Note that atomic vectors are a subset of vectors in R. Vectors can be list or expression as well.

```
is.vector(list())
```

```
[1] TRUE
```

```
is.vector(expression())
```

```
[1] TRUE
```

```
is.atomic(list())
```

```
[1] FALSE
```

```
is.atomic(expression())
```

```
[1] FALSE
```

### 12.0.2.2 Checking the Data Type of an Atomic Vector

To determine the specific type of an atomic vector, use:

- `typeof(x)`: Returns the underlying storage type of `x`.
- `class(x)`: Returns the class of `x`.

```
typeof(numeric_vec)
```

```
[1] "double"
```

```
class(char_vec)
```

```
[1] "character"
```



### 12.0.3 Slicing a vector

An atomic vector can be sliced using the indices of the elements within `[]` brackets. In R

- In R, no indexing backwards - you cannot index from the end
- You can use negative numbers as an index - but they are for removing elements.

For example, considering this vector:

```
num_vec <- c(10, 20, 30, 40, 50)
```

Suppose, we wish to get the 3<sup>rd</sup> element of the vector. We can get it using the index 3:

```
num_vec[3]
```

```
[1] 30
```

A sequence of consecutive elements can be sliced using the indices of the first element and the last element around the `:` operator. For example, let us slice elements from the 1<sup>st</sup> index to the 3<sup>rd</sup> element of the vector `num_vec`:

```
num_vec[1:3]
```

```
[1] 10 20 30
```

We can slice elements at different indices by putting the indices in an atomic vector within the `[]` brackets. Let us slice the 2<sup>nd</sup>, and 4<sup>th</sup> elements of the vector `num_vec`:

```
num_vec[c(2,4)]
```

```
[1] 20 40
```

We can slice consecutive elements, and non-consecutive elements simultaneously. Let us slice the elements from the 1<sup>st</sup> index to the 2<sup>nd</sup> index and the 4<sup>th</sup> element.

```
num_vec[c(1:2,4)]
```

```
[1] 10 20 40
```

### 12.0.3.1 Slicing using a logical atomic vector

An atomic vector can be sliced using a logical atomic vector of the same length. The logical atomic vector will have TRUE values corresponding to the indices where the element is to be selected, and FALSE where the element is to be discarded. See the example below.

```
num_vec[c(TRUE, FALSE, FALSE, TRUE, FALSE)]
```

```
[1] 10 40
```

```
filtered_num_vector <- num_vec[num_vec > 2]
print(filtered_num_vector)
```

```
[1] 10 20 30 40 50
```

### 12.0.4 Mutating a vector

In R, all the data structures are mutable. You can alter and access the content freely.

```
modify an element
num_vec[2] <- 99
print(num_vec)
```

```
[1] 10 99 30 40 50
```

```
adding an element at a specific position
append(num_vec, 3, after=2)
```

```
[1] 10 99 3 30 40 50
```

```
print(num_vec)
```

```
[1] 10 99 30 40 50
```

```
#expanding a vector
new_vec <- c(num_vec, 60, 70)
print(new_vec)
```

```
[1] 10 99 30 40 50 60 70
```

### 12.0.5 Removing elements from a vector

Elements can be removed from the vector using the negative sign within `[]` brackets.

Remove the 2nd element from the vector:

```
new_vec <- new_vec[-3]
print(new_vec)
```

```
[1] 10 99 40 50 60 70
```

If multiple elements need to be removed, the indices of the elements to be removed can be given as an atomic vector.

Remove elements 1 to 3 and element 5 from the vector:

```
new_vec <- new_vec[-c(1:3, 5)]
print(new_vec)
```

```
[1] 50 70
```

### 12.0.6 Iterating

```
vec <- c(10, 20, 30, 40, 50)

for (val in vec) {
 print(val)
}
```

```
[1] 10
[1] 20
[1] 30
[1] 40
[1] 50
```

### 12.0.7 Using as a function input

You can define a function that takes a vector as input and performs operations on its elements.

```
sum_vector <- function(vec) {
 return(sum(vec)) # Sum of all elements
}

Call the function with a vector
my_vec <- c(1, 2, 3, 4, 5)
sum_vector(my_vec)
```

```
[1] 15
```

## 12.0.8 Applying a Function to Each Element of a Vector

If you want to process each element individually, use `sapply()`

```
numbers <- c(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)

Function to check if a number is even or odd
is_even_or_odd <- function(x) {
 if (x %% 2 == 0) {
 return("Even")
 } else {
 return("Odd")
 }
}

Apply the function to each element in the vector
result <- sapply(numbers, is_even_or_odd)

Print the result
print(result)
```

```
[1] "Odd" "Even" "Odd" "Even" "Odd" "Even" "Odd" "Even" "Odd" "Even"
```

Another example:

```
vec <- 1:6
vec
```

```
[1] 1 2 3 4 5 6
```

Suppose, we wish to square each element of the vector. We can use the `sapply()` function as below:

```
sapply(vec, function(x) x**2)
```

```
[1] 1 4 9 16 25 36
```

Note that : `function(x) x**2` is an anonymous function (also called a lambda function) in R.

### 12.0.9 The `rep()` function

The `rep()` function is used to repeat an object a fixed number of times.

```
rep(4, 10)
```

```
[1] 4 4 4 4 4 4 4 4 4 4
```

```
rep(c(2, 3), 10)
```

```
[1] 2 3 2 3 2 3 2 3 2 3 2 3 2 3 2 3 2 3 2 3
```

### 12.0.10 The `which()` function

The `which()` function is used to find the indices of `TRUE` elements in a logical atomic vector.

```
vec <- c(8, 3, 4, 7, 9, 7, 5)
```

```
which(vec == 8)
```

```
[1] 1
```

In the above code, a logical vector is being created with `vec == 8`, and the `which()` function is returning the indices of the `TRUE` elements.

The index of the maximum and minimum values can be found using `which.max()` and `which.min()` respectively. In case of multiple maximum or minimum elements, the smallest index is returned.

```
which.max(vec)
```

```
[1] 5
```

```
which.min(vec)
```

```
[1] 2
```

### 12.0.11 Element-wise operations on atomic vectors

When we use **arithmetic operators** (+, -, \*, /, etc.) or **comparison operators** (>, >=, ==, etc.) between atomic vectors, these operations are applied **element-wise**, meaning they operate on corresponding elements at the **same index** in each vector.

Example: Element-wise Arithmetic Operations

```
vec1 <- c(2, 4, 6)
vec2 <- c(1, 2, 3)

Element-wise addition
result_add <- vec1 + vec2
print(result_add)
```

```
[1] 3 6 9
```

```
Element-wise multiplication
result_mul <- vec1 * vec2
print(result_mul)
```

```
[1] 2 8 18
```

```
Element-wise exponentiation
result_exp <- vec1^vec2
print(result_exp)
```

```
[1] 2 16 216
```

Example: Element-wise Comparison Operations

```
vec1 <- c(10, 20, 30)
vec2 <- c(5, 20, 35)

Element-wise greater than
result_gt <- vec1 > vec2
print(result_gt)
```

```
[1] TRUE FALSE FALSE
```

```
Element-wise equality check
result_eq <- vec1 == vec2
print(result_eq)
```

```
[1] FALSE TRUE FALSE
```

### Recycling Rule in vector operations

If vectors are of different lengths, R applies the recycling rule, where the shorter vector is recycled (repeated) to match the length of the longer vector.

```
vec1 <- c(2, 4, 6, 8)
vec2 <- c(1, 2)

result_recycle <- vec1 + vec2
print(result_recycle)
```

```
[1] 3 6 7 10
```

Operations return a new vector of the same length as the longest input

If an operator is applied between an atomic vector and a scalar, then the operation is performed on each element of the atomic vector and the scalar. See the examples below.

```
vec1*4
```

```
[1] 8 16 24 32
```

```
vec1 > 2
```

```
[1] FALSE TRUE TRUE TRUE
```

Suppose, we wish to slice all elements from the object `vec` that are greater than 2. Here is one approach to do it. We will apply the `>` operator between `vec` and 2 to obtain a logical vector that is `TRUE` on indices where the condition is satisfied, and `FALSE` otherwise. We will then use this logical vector to slice `vec`. Below is the code.

```
vec1[vec1 > 2]
```

```
[1] 4 6 8
```

## 12.0.12 Commonly Used Functions for Atomic Vectors

| Function                   | Description                                | Example                                    |
|----------------------------|--------------------------------------------|--------------------------------------------|
| <code>c()</code>           | Combines elements into a vector            | <code>c(1, 2, 3, 4, 5)</code>              |
| <code>rep(x, times)</code> | Repeats a value multiple times             | <code>rep(5, times=3) → 5 5 5</code>       |
| <code>is.vector(x)</code>  | Checks if an object is a vector            | <code>is.vector(c(1, 2, 3)) → TRUE</code>  |
| <code>length(x)</code>     | Returns the number of elements in a vector | <code>length(c(1, 2, 3)) → 3</code>        |
| <code>unique(x)</code>     | Removes duplicate values                   | <code>unique(c(1, 2, 2, 3)) → 1 2 3</code> |
| <code>sort(x)</code>       | Sorts elements in ascending order          | <code>sort(c(3, 1, 2)) → 1 2 3</code>      |
| <code>rev(x)</code>        | Reverses the order of elements             | <code>rev(c(1, 2, 3)) → 3 2 1</code>       |
| <code>order(x)</code>      | Returns indices that sort the vector       | <code>order(c(10, 5, 8)) → 2 3 1</code>    |

## Mathematical and Statistical Functions

| Function               | Description                          | Example                                |
|------------------------|--------------------------------------|----------------------------------------|
| <code>sum(x)</code>    | Computes the sum of elements         | <code>sum(c(1, 2, 3)) → 6</code>       |
| <code>mean(x)</code>   | Computes the mean (average)          | <code>mean(c(2, 4, 6)) → 4</code>      |
| <code>median(x)</code> | Computes the median                  | <code>median(c(1, 3, 5, 7)) → 4</code> |
| <code>sd(x)</code>     | Computes standard deviation          | <code>sd(c(2, 4, 6, 8)) → 2.58</code>  |
| <code>prod(x)</code>   | Computes the product of all elements | <code>prod(c(2, 3, 4)) → 24</code>     |



| Function              | Description                         | Example                                    |
|-----------------------|-------------------------------------|--------------------------------------------|
| <code>range(x)</code> | Returns the min and max as a vector | <code>range(c(10, 5, 20))</code><br>→ 5 20 |
| <code>min(x)</code>   | Returns the smallest value          | <code>min(c(3, 7, 1))</code> → 1           |
| <code>max(x)</code>   | Returns the largest value           | <code>max(c(3, 7, 1))</code> → 7           |

## Logical and Comparison Functions

| Function                       | Description                                             | Example                                        |
|--------------------------------|---------------------------------------------------------|------------------------------------------------|
| <code>any(x &gt; value)</code> | Checks if any element meets a condition                 | <code>any(c(1, 2, 3) &gt; 2)</code><br>→ TRUE  |
| <code>all(x &gt; value)</code> | Checks if all elements meet a condition                 | <code>all(c(1, 2, 3) &gt; 2)</code><br>→ FALSE |
| <code>which(x == value)</code> | Returns indices of elements equal to <code>value</code> | <code>which(c(5, 10, 5) == 5)</code> → 1 3     |

Next, let's explore how vectorized operations in R improve performance over loops!

## 12.1 Vectorization in R

Vectorization in R refers to the ability to perform operations on entire vectors (or arrays) without the need for explicit loops. This makes computations faster and more efficient because R applies the operation element-wise internally using optimized C or Fortran code.

### 12.1.1 Comparing Loop vs. Vectorized Operations

Let's compare the execution time of a loop-based operation versus a vectorized operation using `system.time()`.

Example: Squaring Each Element in a Vector

Using a `for` Loop (Slow)

```
Create a large vector
vec <- 1:1e6 # 1 to 1,000,000

Squaring using a for loop
square_loop <- function(x) {
 result <- numeric(length(x)) # Pre-allocate memory
```

```

for (i in seq_along(x)) {
 result[i] <- x[i]^2
}
return(result)
}

Measure time taken
system.time(square_loop(vec))

```

```

user system elapsed
0.03 0.00 0.03

```

Using Vectorized Operation (Fast)

```

Squaring using vectorized approach
square_vectorized <- function(x) {
 return(x^2) # Directly applies to the whole vector
}

Measure time taken
system.time(square_vectorized(vec))

```

```

user system elapsed
0 0 0

```

Key takeaway:

- The `for` loop takes significantly longer (because it iterates element-by-element).
- The vectorized operation is much faster (because R applies the operation to the entire vector at once).

### 12.1.2 Performance Comparison: Loop vs. `sapply()` vs. Vectorized

Example: Calculating square roots

```

Create a large vector
x <- 1:1000000

Using a for loop
system.time({

```

```

result_loop <- numeric(length(x))
for(i in seq_along(x)) {
 result_loop[i] <- sqrt(x[i])
}
})

```

```

user system elapsed
0.01 0.00 0.01

```

```

Using sapply
system.time({
 result_sapply <- sapply(x, sqrt)
})

```

```

user system elapsed
0.37 0.02 0.40

```

```

Using vectorized operation (for comparison)
system.time({
 result_vector <- sqrt(x)
})

```

```

user system elapsed
0 0 0

```

- for loop and ‘sapply()’ performs comparably.
- The direct vectorized operation (`sqrt(vec)`) is the fastest.

### 12.1.3 Functions that support vectorization

Many R functions are already vectorized, such as:

| Operation             | Function                                                                                                  |
|-----------------------|-----------------------------------------------------------------------------------------------------------|
| <b>Arithmetic</b>     | <code>+</code> , <code>-</code> , <code>*</code> , <code>/</code> , <code>^</code>                        |
| <b>Logical</b>        | <code>&gt;</code> , <code>&lt;</code> , <code>==</code> , <code>!=</code>                                 |
| <b>Math Functions</b> | <code>sqrt()</code> , <code>log()</code> , <code>exp()</code> , <code>abs()</code> , <code>round()</code> |
| <b>Aggregation</b>    | <code>sum()</code> , <code>mean()</code> , <code>max()</code> , <code>min()</code> , <code>sd()</code>    |

### 12.1.4 Understanding why vectorization is faster

Reasons Vectorized Operations are More Efficient:

- R is optimized for vectorized operations (implemented in C under the hood).
- No explicit loops are needed, reducing computation overhead.
- Memory allocation is handled efficiently, unlike loops where repeated memory allocation slows execution.
- Parallel execution is possible for some vectorized functions.

In R, both vectors and matrices support full vectorization, meaning element-wise operations can be applied directly without explicit loops.

## 12.2 Practice exercises

### 12.2.1 Exercise 1

USA's GDP per capita from 1960 to 2021 is given by the vector `G` in the code chunk below. The values are arranged in ascending order of the year, i.e., the first value is for 1960, the second value is for 1961, and so on.

```
G = c(3007, 3067, 3244, 3375, 3574, 3828, 4146, 4336, 4696, 5032, 5234, 5609, 6094, 6726, 7226, 7800)
```

#### 12.2.1.1

What was the GDP per capita in 1990?

#### 12.2.1.2

What is the mean, median, and standard deviation of the GDP per capita?

#### 12.2.1.3

Find the first year where GDP per capita exceeded \$50,000.

#### 12.2.1.4

What is the range of GDP per capita over the years?

### 12.2.1.5

In which years did the GDP per capita increase by more than 10%?

Here is the loop version

```
years <- c()
for (i in 1:(length(G) - 1)) {
 diff <- (G[i+1] - G[i]) / G[i]
 if (diff > 0.1) years <- c(years, 1960 + i)
}
print(years)
```

```
[1] 1973 1976 1977 1978 1979 1981 1984
```

You need to solve this problem without using a `for` loop.

### 12.2.1.6

Find the year with the highest GDP per capita growth (percentage-wise).

### 12.2.1.7

Identify all years where GDP per capita decreased.

## 12.2.2 Exercise 2

Below is a vector consisting of responses to the question: “At what age do you think you will marry?” from students of the STAT303-1 Fall 2022 class.

```
exp_marriage_age <- c('24', '30', '28', '29', '30', '27', '26', '28', '30+', '26', '28', '30', '30', '30'
```

### 12.2.2.1 Cleaning data

Remove the elements that are not integers - such as ‘probably never’, ‘30+’, etc. Convert the remaining elements to integer. What is the length of the new vector?

Hint: Use `is.na()` to detect missing values.

### 12.2.2.2 Capping unreasonably high values

Cap the values greater than 80 to 80, in the clean vector obtained above. What is the mean age when people expect to marry in the new vector?

### 12.2.2.3 People marrying at 30 or more

### 12.2.3 Exercise 3

Write a function that identifies if a word is a **palindrome** (*A palindrome is a word that reads the same both backwards and forwards, for example, peep, rotator, madam, etc.*). Apply the function to the vector of words below to count the number of palindrome words.

```
words_vec <- c('fat', 'civic', 'radar', 'mountain', 'noon', 'papa')
```

```
palindrome <- function(word) {
 for (i in 1:as.integer(nchar(word)/2)) {
 if (substr(word, i, i) != substr(word, nchar(word) - (i-1), nchar(word) - (i-1))) {
 return(FALSE)
 }
 }
 return(TRUE)
}
sum(sapply(words_vec, palindrome))
```

```
[1] 3
```

Try to solve this without using a for loop

Hints:

- Define a function to **reverse a word**. You may need `strsplit()` to split the word into individual characters and `paste()` to concatenate them back into a word.
- Use `sapply()` to apply a function to each element of an **atomic vector**.

### Understanding collapse vs. sep in paste() in R

The `paste()` function in R allows you to **combine strings**, but `collapse` and `sep` have different roles.

### sep (Separator Between Arguments)

- Used when **multiple arguments** are passed to `paste()`.
- Defines the **separator between these arguments**.

```
paste("Hello", "World", sep = "-")
```

```
[1] "Hello-World"
```

```
paste("R", "is", "fun", sep = " | ")
```

```
[1] "R | is | fun"
```

Use `sep` when combining multiple elements passed separately.

### collapse (Collapsing a Vector into One String)

- Used when concatenating elements of a vector into a single string.
- Defines the separator between vector elements.

```
char_vec <- c("R", "is", "fun")
paste(char_vec, collapse = " ")
```

```
[1] "R is fun"
```

```
char_vec <- c("apple", "banana", "cherry")
paste(char_vec, collapse = ", ")
```

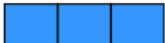


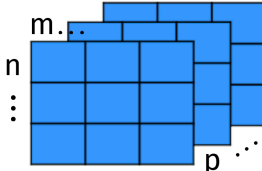
```
[1] "apple, banana, cherry"
```

```
char_vec <- c("l", "o", "v", "e")
paste(char_vec, collapse = "")
```

```
[1] "love"
```

## 13 R: Data structures

R has several **data structures** to store and manipulate data efficiently. These structures can be classified into **homogeneous** (same type) and **heterogeneous** (different types) categories.

|                   | Dimensions                                                                                                          | Mode (data "type") | Example                                                                 |
|-------------------|---------------------------------------------------------------------------------------------------------------------|--------------------|-------------------------------------------------------------------------|
| <b>Vector</b>     | 1                                  | Identical          | <code>c(10,0.2,34,48,53)</code>                                         |
| <b>Matrix</b>     |                                    | Identical          | <code>matrix(c(1,2,3,<br/>11,12,13),<br/>nrow = 2,<br/>ncol = 3)</code> |
| <b>Data frame</b> |                                   | Can be different   | <code>data.frame(x = 1:3,<br/>y = 5:7)</code>                           |
| <b>Array</b>      |                                  | Identical          | <code>array(data = 1:3,<br/>dim = c(2,4,2))</code>                      |
| <b>List</b>       | $\left\{ \begin{array}{l} \text{Vector} \\ \text{Matrix} \\ \text{Data frame} \\ \text{Array} \end{array} \right\}$ | Can be different   | <code>list(x = cars[,1],<br/>y = cars[,2])</code>                       |



## 13.1 Atomic vectors ((1D, Homogeneous))

**Atomic vectors** are the most fundamental data type in R. They are **one-dimensional collections of elements**, where all elements must share the **same data type**.

Since atomic vectors were covered in the previous chapter, this chapter will focus on the remaining three data structures: **matrices, lists, and data frames**.

## 13.2 Matrix ((2D, Homogeneous))

Matrices are two-dimensional arrays.

### 13.2.1 Creating a matrix

The built-in function `matrix()` is used to define a matrix. An atomic vector can be organized as a matrix by specifying the number of rows and columns.

For example, let us define a 3x3 matrix (3 rows and 3 columns) consisting of consecutive integers from 1 to 9.

```
mat <- matrix(1:9, 3, 3)
mat
```

|      | [,1] | [,2] | [,3] |
|------|------|------|------|
| [1,] | 1    | 4    | 7    |
| [2,] | 2    | 5    | 8    |
| [3,] | 3    | 6    | 9    |

Note that the integers fill up column-wise in the matrix. If we wish to fill-up the matrix by row, we can use the `byrow` argument.

```
mat <- matrix(1:9, 3, 3, byrow = TRUE)
mat
```

|      | [,1] | [,2] | [,3] |
|------|------|------|------|
| [1,] | 1    | 2    | 3    |
| [2,] | 4    | 5    | 6    |
| [3,] | 7    | 8    | 9    |

### 13.2.2 Getting matrix dimensions, the number of rows, the number of columns

Get matrix dimensions will give vector of c(rows, cols): - `dim()`

```
dim(mat)
```

```
[1] 3 3
```

Get number of elements: `length()`

```
length(mat)
```

```
[1] 9
```

The functions `nrow()` and `ncol()` can be used to get the number of rows and columns of the matrix respectively.

```
nrow(mat)
```

```
[1] 3
```

```
ncol(mat)
```

```
[1] 3
```

### 13.2.3 Matrix indexing and Slicing

Matrices can be sliced using the indices of row and column separated by a `,` in box brackets. Suppose we wish to get the element in the  $2^{nd}$  row and  $3^{rd}$  column of the matrix:

```
mat[2, 3]
```

```
[1] 6
```

For selecting all rows or columns of a matrix, the index for the row/column can be left blank. Suppose we wish to get all the elements of the  $1^{st}$  of the matrix:

```
mat[1,]
```

```
[1] 1 2 3
```

Row and columns of the matrix can be sliced using the `:` operator. Suppose we want to select a sub-matrix that has elements in the first two rows and columns 2 and 3 of the matrix `mat`:

```
mat[1:2, 2:3]
```

```
 [,1] [,2]
[1,] 2 3
[2,] 5 6
```

### 13.2.4 Working with `rownames()` and `colnames()`

In R, row names (`rownames()`) and column names (`colnames()`) allow you to label and reference rows and columns in matrices and data frames efficiently.

```
Create a matrix
mat <- matrix(1:9, nrow = 3, byrow = TRUE)

Assign row names
rownames(mat) <- c("Row1", "Row2", "Row3")

Assign column names
colnames(mat) <- c("Col1", "Col2", "Col3")

Print matrix
print(mat)
```

```
 Col1 Col2 Col3
Row1 1 2 3
Row2 4 5 6
Row3 7 8 9
```

Getting and Modifying Names

```
rownames(mat) # Get row names
```

```
[1] "Row1" "Row2" "Row3"
```

```
colnames(mat) # Get column names
```

```
[1] "Col1" "Col2" "Col3"
```

```
rownames(mat) <- c("A", "B", "C") # Modify row names
colnames(mat) <- c("X", "Y", "Z") # Modify column names

print(mat)
```

```
 X Y Z
A 1 2 3
B 4 5 6
C 7 8 9
```

### 13.2.5 Adding and Removing

- use `cbind()` to add a column of correct length
- use `rbind()` to add a row of correct length
- if length is incorrect there will be a warning (no error it will still run), however it will “recycle” values.
- When adding a row or a col, you CANNOT insert it between the existing rows/cols. However, you can re-arrange AFTER it is added using indexing.
- use negative indexes to remove rows/columns

```
new_col <- c(7, 8, 9)

add the column
mat_new <- cbind(mat, new_col)
print(mat_new)
```

```
 X Y Z new_col
A 1 2 3 7
B 4 5 6 8
C 7 8 9 9
```

```
mat_new <- cbind(mat, new_col = 5)
print(mat_new)
```

```
 X Y Z new_col
A 1 2 3 5
B 4 5 6 5
C 7 8 9 5
```

```
new_row <- c(7, 8, 9)

add the row
mat_new <- rbind(mat, new_row)
print(mat_new)
```

```
 X Y Z
A 1 2 3
B 4 5 6
C 7 8 9
new_row 7 8 9
```

```
mat_new <- rbind(mat, new_row = 5)
print(mat_new)
```

```
 X Y Z
A 1 2 3
B 4 5 6
C 7 8 9
new_row 5 5 5
```

```
remove the added row
mat_new <- mat_new[-4,]
print(mat_new)
```

```
 X Y Z
A 1 2 3
B 4 5 6
C 7 8 9
```

```
remove the added column
mat_new <- mat_new[, -4]
print(mat_new)
```

```
 X Y Z
A 1 2 3
B 4 5 6
C 7 8 9
```

### 13.2.6 Iterating

To iterate through each element you generally need a nested loop combined with `nrow()` and `ncol()`

```
for (i in 1: nrow(mat_new)){
 for (j in 1: ncol(mat_new)) {
 print(paste("Element at (", i, ", ", j, "):", mat[i, j]))
 }
}
```

```
[1] "Element at (1 , 1): 1"
[1] "Element at (1 , 2): 2"
[1] "Element at (1 , 3): 3"
[1] "Element at (2 , 1): 4"
[1] "Element at (2 , 2): 5"
[1] "Element at (2 , 3): 6"
[1] "Element at (3 , 1): 7"
[1] "Element at (3 , 2): 8"
[1] "Element at (3 , 3): 9"
```

### 13.2.7 Element-wise arithmetic operations

Element-wise arithmetic operations have the same logic with atomic vectors, they can be performed between 2 matrices of the same shape.

```
mat1 <- matrix(1:6, 2, 3)
mat2 <- matrix(c(9, 2, 6, 5, 1, 0), 2, 3)
mat1 + mat2
```

|      | [,1] | [,2] | [,3] |
|------|------|------|------|
| [1,] | 10   | 9    | 6    |
| [2,] | 4    | 9    | 6    |

```
mat1 - mat2
```

|      | [,1] | [,2] | [,3] |
|------|------|------|------|
| [1,] | -8   | -3   | 4    |
| [2,] | 0    | -1   | 6    |

Suppose we need to sum up all the rows of the matrix. We can do it using a **for** loop as follows:

```
row_sum <- c(0,0)
for (i in 1:nrow(mat)) {
 for (j in 1:ncol(mat)) {
 row_sum[i] <- row_sum[i] + mat[i, j]
 }
}
row_sum
```

```
[1] 6 15 NA
```

Observe that in the above **for** loop, elements of each row are added one at a time. We can add all the elements of a row simultaneously using the **sum()** function. This will reduce a **for** loop from the above code:

```
row_sum <- c(0,0)
for (i in 1:nrow(mat)){
 row_sum[i] <- sum(mat[i,])
}
row_sum
```

```
[1] 6 15 24
```

In the above code, we sum up all the elements of the row simultaneously. However, we still need to sum up the elements of each row one at a time.

### 13.2.7.1 Matrix multiplication vs. element-wise multiplication

- Matrix multiplication: using `%%`

```
Create another 3x3 matrix
mat2 <- matrix(9:1, nrow = 3)

Perform matrix multiplication
result <- mat %% mat2
print(result)
```

|   | [,1] | [,2] | [,3] |
|---|------|------|------|
| A | 46   | 28   | 10   |
| B | 118  | 73   | 28   |
| C | 190  | 118  | 46   |

- Element-wise multiplication: using `*`

```
Element-wise multiplication
mat * mat
```

|   | X  | Y  | Z  |
|---|----|----|----|
| A | 1  | 4  | 9  |
| B | 16 | 25 | 36 |
| C | 49 | 64 | 81 |

### 13.2.8 The `apply()` function

The `apply()` function can be used to apply a function on each row or column of a matrix. Thus, this function helps avoid the user to write a `for()` loop in R to iterate over all the rows and columns of the matrix. Note that each row / column of a matrix is an atomic vector. Thus, vectorized computations can be performed within the function, resulting in efficient computations.

Note that the `apply` functions use a `for()` loop under-the-hood, and thus the function will be applied sequentially on each row / column of the matrix. However, as the implementation of the `for()` loop is in C, it is likely to be faster than a `for()` loop in R.

Let us use the `apply()` function to sum up all the rows of the matrix `mat`.



```
apply(mat, 1, sum)
```

```
A B C
6 15 24
```

Let us compare the time taken to sum up rows of a matrix using a `for` loop with the time taken using the `apply()` function.

```
options(digits.secs = 6)
start.time <- Sys.time()
row_sum<-c(0, 0)
for (i in 1:nrow(mat)){
 row_sum[i] <- sum(mat[i,])
}
row_sum
```

```
[1] 6 15 24
```

```
end.time <- Sys.time()
time.taken <- end.time - start.time
time.taken
```

Time difference of 0.00346899 secs

```
start.time <- Sys.time()
apply(mat, 1, sum)
```

```
A B C
6 15 24
```

```
end.time <- Sys.time()
time.taken <- end.time - start.time
time.taken
```

Time difference of 0.001055956 secs

Observe that the `apply()` function takes much lesser time to sum up all the rows of the matrix as compared to the `for` loop.

### 13.2.9 Misc useful functions:

Check if a value/string is in the matrix:

- `"val" %in% my_mat`

```
Create a matrix
fruit_mat <- matrix(c("apple", "banana", "cherry", "date"), nrow=2, byrow=TRUE)

Check if "banana" is in the matrix
"banana" %in% fruit_mat
```

```
[1] TRUE
```

```
Check if "grape" is in the matrix
"grape" %in% fruit_mat
```

```
[1] FALSE
```

If you want to return TRUE or FALSE directly:

```
any(fruit_mat == "banana")
```

```
[1] TRUE
```

```
any(fruit_mat == "grape")
```

```
[1] FALSE
```

```
which(mat == "banana", arr.ind = TRUE)
```

```
row col
```

Check if a value is missing (ie: NA):

- `is.na(val)`

## 13.3 changes

## 13.4 Lists (1D, Heterogeneous)

Atomic vectors and matrices are quite useful in R. However, a constraint with them is that they can only contain objects of the same datatype. For example, an atomic vector can contain all numeric objects, all character objects, or all logical objects, but not a mixture of multiple types of objects. Thus, there arises a need for a `list` data structure that can store objects of multiple datatypes.

### 13.4.1 Creating and Indexing

A list can be defined using the `list()` function. For example, consider the list below:

```
list_ex <- list(1, "apple", TRUE, list("another list", TRUE))
```

The list `list_ex` consists of objects of multiple datatypes. The length of the list can be obtained using the `length()` function:

```
length(list_ex)
```

```
[1] 4
```

A list is an ordered collection of objects. Each object of the list is associated with an index that corresponds to its order of occurrence in the list.

A single element can be sliced from the list by specifying its index within the `[[ ]]` operator. Let us slice the 2<sup>nd</sup> element of the list `list_ex`:

```
list_ex[[2]]
```

```
[1] "apple"
```

Multiple elements can be sliced from the list by specifying the indices as an atomic vector within the `[ ]` operator. Let us slice the 1<sup>st</sup> and 3<sup>rd</sup> elements from the list `list_ex`:

```
list_ex[c(1,3)]
```

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

[[2]]
[1] TRUE
```

### 13.4.2 Naming a list

Elements of a list can be named using the `names()` function. Let us name the elements of `list_ex`:

```
names(list_ex) <- c("Name1", "second_name", "3rd_element", "Number 4")
```

A single element can be sliced from the list using the name of the element with the `$` operator. Let us slice the element named as `second_name` from the list `list_ex`:

```
list_ex$second_name
```

```
[1] "apple"
```

Note that if the name of the element does not begin with a letter or has special characters such as a space, then it should be specified within single quotes after the `$` operator. For example, let us slice the element named as `3rd_element` from the list `list_ex`:

```
list_ex$`3rd_element`
```

```
[1] TRUE
```

Names of elements of a list can also be specified while defining the list, as in the example below:

```
list_ex_with_names <- list(movie = 'The Dark Knight', IMDB_rating = 9)
```

### 13.4.3 Using as Function Input

No arithmetic or logical operations with a list, since elements are with different data types. A list can be converted to an atomic vector using the `unlist()` function. For example, let us convert the list `list_ex` to a vector:

```
unlist(list_ex)
```

| Name1 | second_name | 3rd_element | Number 41      | Number 42 |
|-------|-------------|-------------|----------------|-----------|
| "1"   | "apple"     | "TRUE"      | "another list" | "TRUE"    |

Since a vector can contain objects of a single datatype, note that all objects have been converted to the **character** datatype in the vector above.

Another example:

```
Define a list with mixed data types
my_list <- list(a = 1, b = 2, c = "text", d = 4, e = TRUE)

Function to sum numeric elements in the list
sum_list_numeric <- function(lst) {
 # Convert list to a vector, forcing non-numeric elements to NA
 vec <- unlist(lst)

 # Keep only numeric values (ignoring non-numeric ones)
 numeric_values <- as.numeric(vec[!is.na(as.numeric(vec))])

 # Compute sum
 return(sum(numeric_values))
}

Call the function
result <- sum_list_numeric(my_list)
```

Warning in sum\_list\_numeric(my\_list): NAs introduced by coercion

```
print(result)
```

```
[1] 7
```

#### 13.4.4 Applying a function to each element of a list: the `lapply()` function

```
Define a list with numeric values
num_list <- list(a = 1:5, b = 6:10, c = 11:15)

Use lapply() to compute the sum of each list element
sum_results <- lapply(num_list, sum)

Print the result
print(sum_results)
```

```
$a
[1] 15
```

```
$b
[1] 40
```

```
$c
[1] 65
```

- Using lapply() with an Anonymous Function

```
Compute the mean of each element in the list
mean_results <- lapply(num_list, function(x) mean(x))

Print the result
print(mean_results)
```

```
$a
[1] 3
```

```
$b
[1] 8
```

```
$c
[1] 13
```

## 13.5 Data Frames (2D, Heterogeneous)

A data frame is a table-like structure where each column is a vector. Unlike matrices, columns can have different types.

### 13.5.1 Creating data Frames

```
df <- data.frame(Name=c("Alice", "Bob", "Charlie"),
 Age=c(25, 30, 35),
 Score=c(90, 85, 88))

print(df)
```

```
 Name Age Score
1 Alice 25 90
2 Bob 30 85
3 Charlie 35 88
```

### 13.5.2 Common functions

| Function     | Purpose                                                                     |
|--------------|-----------------------------------------------------------------------------|
| nrow(df)     | Number of rows                                                              |
| ncol(df)     | Number of columns                                                           |
| dim(df)      | Returns (rows, columns) as a vector                                         |
| length(df)   | Number of columns (since a data frame is a list of columns)                 |
| rownames(df) | Returns row names                                                           |
| colnames(df) | Returns column names                                                        |
| str(df)      | Displays the structure of the data frame, including column types and values |
| head(df)     | Returns the first six rows of the data frame                                |

### 13.5.3 Missing values

#### 13.5.3.1 Using is.na()

```
Sample data frame with missing values
df_missing <- data.frame(Name = c("Alice", "Bob", "Charlie"),
 Age = c(25, NA, 35),
 Score = c(90, 85, NA))

Check for missing values (returns TRUE/FALSE matrix)
is.na(df_missing)
```

```

 Name Age Score
[1,] FALSE FALSE FALSE
[2,] FALSE TRUE FALSE
[3,] FALSE FALSE TRUE

```

`is.na(df)` returns a matrix where TRUE indicates a missing value.

### 13.5.3.2 Count the number of total missing values

```
sum(is.na(df))
```

```
[1] 0
```

Counts all NA values in the entire data frame.

### 13.5.3.3 Count missing values by column

```
colSums(is.na(df))
```

```

Name Age Score
 0 0 0

```

### 13.5.3.4 Find Rows Containing NA Values

```
df[!complete.cases(df),]
```

```

[1] Name Age Score
<0 rows> (or 0-length row.names)

```

Common functions on missing values

| Function                    | Purpose                                           |
|-----------------------------|---------------------------------------------------|
| <code>is.na(df)</code>      | Returns a matrix of TRUE/FALSE for missing values |
| <code>sum(is.na(df))</code> | Counts the total number of missing values         |



| Function                               | Purpose                                     |
|----------------------------------------|---------------------------------------------|
| <code>colSums(is.na(df))</code>        | Counts missing values per column            |
| <code>df[!complete.cases(df), ]</code> | Shows only rows that contain missing values |
| <code>any(is.na(df))</code>            | Returns TRUE if any missing values exist    |
| <code>mean(is.na(df))</code>           | Returns the proportion of missing values    |

### 13.5.4 Accessing Data Frame Elements

```
df$Name # Accessing a column
```

```
[1] "Alice" "Bob" "Charlie"
```

```
df[2,] # Second row
```

```
 Name Age Score
2 Bob 30 85
```

```
df[, "Age"] # Age column
```

```
[1] 25 30 35
```

```
df[1:2, c("Name", "Score")] # Subset
```

```
 Name Score
1 Alice 90
2 Bob 85
```

### 13.5.5 Using Logical Operators for Data Frame Subsetting

#### 13.5.5.1 Using Logical Operators ([ ])

You can filter rows based on conditions using comparison operators (`==`, `>`, `<`, `>=`, `<=`, `!=`):

Filter rows where age is greater than 25

```
filtered_df <- df[df$Score == 90,]
print(filtered_df)
```

|   | Name  | Age | Score |
|---|-------|-----|-------|
| 1 | Alice | 25  | 90    |

```
filtered_df <- df[df$Age > 25,]
print(filtered_df)
```

|   | Name    | Age | Score |
|---|---------|-----|-------|
| 2 | Bob     | 30  | 85    |
| 3 | Charlie | 35  | 88    |

- `df$Age > 25` creates a logical vector (`FALSE`, `TRUE`, `TRUE`).
- `df[df$Age > 25, ]` selects rows where `Age` is greater than 25.
- Uses `==` to filter rows where `Score` is exactly 90.

#### 13.5.5.2 Using `subset()` function

```
filtered_df <- subset(df, Age > 25)
print(filtered_df)
```

|   | Name    | Age | Score |
|---|---------|-----|-------|
| 2 | Bob     | 30  | 85    |
| 3 | Charlie | 35  | 88    |

#### 13.5.5.3 Filter using multiple conditions

You can combine multiple conditions with `&` (AND) and `|` (OR):

```
filtered_df <- df[df$Age > 25 & df$Score > 85,]
print(filtered_df)
```

|   | Name    | Age | Score |
|---|---------|-----|-------|
| 3 | Charlie | 35  | 88    |

Using `subset()`

```
filtered_df <- subset(df, Age > 28 & Score > 85)
print(filtered_df)
```

```
 Name Age Score
3 Charlie 35 88
```

```
filtered_df <- df[df$Age < 30 | df$Score > 85,]
print(filtered_df)
```

```
 Name Age Score
1 Alice 25 90
3 Charlie 35 88
```

```
Filter and select only the "Name" and "Score" columns
filtered_df <- subset(df, Age > 28 & Score > 85, select = c(Name, Score))
print(filtered_df)
```

```
 Name Score
3 Charlie 88
```

### 13.5.6 Iterating

```
Iterate over rows
for (i in 1:nrow(df)) {
 print(paste("Row", i, ":", df[i,]))
}
```

```
[1] "Row 1 : Alice" "Row 1 : 25" "Row 1 : 90"
[1] "Row 2 : Bob" "Row 2 : 30" "Row 2 : 85"
[1] "Row 3 : Charlie" "Row 3 : 35" "Row 3 : 88"
```

```
for (col in names(df)) {
 print(paste("Column:", col))
 print(df[[col]]) # Access column by name
}
```

```
[1] "Column: Name"
[1] "Alice" "Bob" "Charlie"
[1] "Column: Age"
[1] 25 30 35
[1] "Column: Score"
[1] 90 85 88
```

```
for (i in 1:nrow(df)) {
 for (j in 1:ncol(df)) {
 print(paste("Element at [", i, ",", j, "]:", df[i, j]))
 }
}
```

```
[1] "Element at [1 , 1]: Alice"
[1] "Element at [1 , 2]: 25"
[1] "Element at [1 , 3]: 90"
[1] "Element at [2 , 1]: Bob"
[1] "Element at [2 , 2]: 30"
[1] "Element at [2 , 3]: 85"
[1] "Element at [3 , 1]: Charlie"
[1] "Element at [3 , 2]: 35"
[1] "Element at [3 , 3]: 88"
```

## 13.6 Summary

| Data Structure       | Dimensions | Homogeneous? | Example                                 |
|----------------------|------------|--------------|-----------------------------------------|
| <b>Atomic Vector</b> | 1D         | Yes          | <code>c(1, 2, 3)</code>                 |
| <b>List</b>          | 1D         | No           | <code>list(name="Alice", age=25)</code> |
| <b>Matrix</b>        | 2D         | Yes          | <code>matrix(1:6, nrow=2)</code>        |
| <b>Data Frame</b>    | 2D         | No           | <code>data.frame(name, age)</code>      |

## 13.7 Practice exercises

### 13.7.1 Exercise 1

Recall the earlier example where we computed year's in which the increase in GDP per capita was more than 10%.

### 13.7.1.1

Let us use matrices to solve the problem. We'll also compare the time it takes using a matrix with the time it takes using for loops.

```
G = c(3007, 3067, 3244, 3375, 3574, 3828, 4146, 4336, 4696, 5032, 5234, 5609, 6094, 6726, 7226, 7800)

start.time <- Sys.time()

#Let the first column of the matrix be the GDP of all the years except 1960, and the second column be the GDP of 1960
GDP_mat <- matrix(c(G[-1], G[-length(G)]), length(G) - 1, 2)

#The percent increase in GDP can be computed by performing computations using the 2 columns of the matrix
inc <- (GDP_mat[,1] - GDP_mat[,2]) / GDP_mat[,2]
years <- 1961:2021
years <- years[inc > 0.1]
years
```

```
[1] 1973 1976 1977 1978 1979 1981 1984
```

```
end.time <- Sys.time()
time.taken <- end.time - start.time
time.taken
```

Time difference of 0.002213001 secs

Without matrices, the time taken to perform the same computation is measured with the code below.

```
start.time <- Sys.time()
years <- c()
for (i in 1:(length(G) - 1)) {
 diff = (G[i+1] - G[i]) / G[i]
 if (diff > 0.1) years <- c(years, 1960 + i)
}
print(years)
```

```
[1] 1973 1976 1977 1978 1979 1981 1984
```

```
#print(proc.time()[3]-start_time)
end.time <- Sys.time()
time.taken <- end.time - start.time
time.taken
```

Time difference of 0.005369902 secs

Observe that matrices reduce the execution time of the code as computations are performed using vectorization, in contrast to a `for` loop where computations are performed one at a time.

### 13.7.1.2

Determine the highest GDP per capita in the United States for each five-year period from 1961-1965 to 2015-2020.

**Hint:**

- First, create a matrix where each row represents a five-year period of GDP per capita.
- Then, use the `apply()` function to find the maximum GDP per capita within each period

```
GDP_5year <- matrix(G[-c(1, length(G))], 12, 5, byrow = TRUE)
GDP_max_5year <- apply(GDP_5year, 1, max)
```

In the above code, we applied the built-in function `max` to each row (5-year period).

### 13.7.1.3

Identify the highest GDP per capita in the United States across all the five-year periods defined in the previous question. Additionally, determine the specific five-year period and the exact year in which this maximum GDP per capita occurred.

```
Define start year
start_year <- 1961

Generate period labels
year_intervals <- seq(start_year, start_year + (nrow(GDP_5year) - 1) * 5, by = 5)
period_labels <- paste(year_intervals, year_intervals + 4, sep = "-")

Assign row names
```

```

rownames(GDP_5year) <- period_labels

Find the maximum GDP across all 5-year periods
max_gdp_overall <- max(GDP_5year)

Find the row and column index of the maximum value
max_position <- which(GDP_5year == max_gdp_overall, arr.ind = TRUE)

Extract the corresponding 5-year period
max_period <- rownames(GDP_5year)[max_position[1]]

Compute the exact year within the period
max_year_within_period <- start_year + (max_position[1] - 1) * 5 + (max_position[2] - 1)

Print results
print(paste("Maximum GDP per capita:", max_gdp_overall))

```

```
[1] "Maximum GDP per capita: 65095"
```

```
print(paste("Occurred in period:", max_period))
```

```
[1] "Occurred in period: 2016-2020"
```

```
print(paste("Occurred in year:", max_year_within_period))
```

```
[1] "Occurred in year: 2019"
```

If a built-in function is not available for the required computations, we can create a user-defined function and pass it to the `apply()` function to perform the operation efficiently. See example below

#### 13.7.1.4

Identify the five-year periods which the GDP per capita experienced a continuous decline each year, indicating a period of financial crisis or economic degradation

```
Define a function to check if all values in a row are decreasing
is_decreasing <- function(row) {
 return(all(diff(row) < 0))
}

Identify periods where GDP decreases year by year
decreasing_periods <- apply(GDP_5year, 1, is_decreasing)

Extract period names where GDP is continuously decreasing
decreasing_period_names <- rownames(GDP_5year)[decreasing_periods]

Print the periods of financial decline
print(decreasing_period_names)
```

character(0)

Instead of defining a named function, you can also use an anonymous function directly within the `apply()` function for a more concise approach

```
Apply a user-defined function to check for decreasing GDP in each period
decreasing_periods <- apply(GDP_5year, 1, function(row) all(diff(row) < 0))

Extract the period names where GDP decreased every year
decreasing_period_names <- rownames(GDP_5year)[decreasing_periods]

Print the periods of financial decline
print(decreasing_period_names)
```

character(0)

The 5 year periods during which the GDP per capita decreased as compared to the previous year are 2006-2010, and 2016-2020.

### 13.7.2 Exercise 2

Download the file *capital\_cities.csv* from [here](#). Make sure the file is in your current working directory. Execute the following code to obtain the objects `coordinates_capital_cities` and `country_names`.

The object `country_names` is an atomic vector consisting of country names. The object `coordinates_capital_cities` is a matrix consisting of the latitude-longitude pair of the



capital city of the respective country. The order of countries in `country_names` is the same as the order in which their capital city coordinates (latitude-longitude) appear in the matrix `coordinates_capital_cities`.

```
capital_cities <- read.csv('capital_cities.csv')
coordinates_capital_cities <- as.matrix(capital_cities[,c(3, 4)])
country_names <- capital_cities[,1]
```

### 13.7.2.1 Top 10 countries closest to DC

1. Print the names of the countries of the top 10 capital cities closest to the US capital - Washington DC.
2. Create and print a matrix containing the coordinates of the top 10 capital cities closest to Washington DC.

Note that:

1. The *Country Name* for US is given as *United States* in the data.
2. The ‘closeness’ of capital cities from the US capital is based on the Euclidean distance of their coordinates to those of the US capital.
3. The dataset includes two records: *United States* and *US Minor Outlying Islands*, both belonging to the U.S., with a distance of zero. These records must be excluded to accurately identify the closest foreign capital cities.

#### Hint:

1. Get the coordinates of Washington DC from `coordinates_capital_cities`. The row that contains the coordinates of DC will have the same index as `United States` has in the vector `country_names`
2. Create a matrix that has coordinates of Washington DC in each row, and has the same number of rows as the matrix `coordinates_capital_cities`.
3. Subtract `coordinates_capital_cities` from the matrix created in (2). Element-wise subtraction will occur between the matrices.
4. Use the `apply()` function on the matrix obtained above to find the Euclidean distance of Washington DC from the rest of the capital cities.
5. Using the distances obtained above, find the country that has the closest capital to DC (Euclidean distance > 0).

```

Find the index of the US capital (Washington, D.C.)
US_index = which(country_names == 'United States')
dc_coord <- coordinates_capital_cities[US_index,]

Compute Euclidean distances to Washington, D.C.
distances_to_DC <- apply(coordinates_capital_cities, 1,
 function(city_coord) sqrt(sum((city_coord - dc_coord)**2)))

Create a matrix with country index and distance
num_of_countries <- length(country_names)
distances_to_DC_matrix <- cbind(1:num_of_countries, distances_to_DC)

Exclude records where the distance is exactly 0
distances_to_DC_matrix <- distances_to_DC_matrix[distances_to_DC_matrix[, 2] > 0,]

Sort by distance
sorted <- distances_to_DC_matrix[order(distances_to_DC_matrix[,2]),]

```

Top 10 countries with capitals closest to Washington DC are the following:

```

Get the top 10 closest countries (after exclusions)
top_10_countries <- sorted[1:10, 1]

Print the country name
print(country_names[sorted[1:10, 1]])

```

You can also use the following code to exclude the U.S. records

```

#Exclude rows where the country is "United States" or "US Minor Outlying Islands"
filtered_sorted <- sorted[!(country_names[sorted[, 1]] %in% c("United States", "US Minor Outlying Islands")),]

```

The coordinates of the top 10 capital cities closest to Washington DC are:

```

coordinates_capital_cities[top_10_countries,]

```

### 13.7.2.2 Use data.frame to solve the above task

### 13.7.3 Exercise 3

Download the dataset *movies.csv*. Execute the following code to read the data into the object `movies`:

```
movies <- read.csv("movies.csv", stringsAsFactors = FALSE)
```

### 13.7.3.1

What is the datatype of the object `movies`?

```
class(movies)
```

The datatype of the object `movies` is list.

### 13.7.3.2

Count the number movies having a negative profit, i.e., their production budget is higher than their worldwide gross.

Ignore the movies that:

1. Have missing values of production budget or worldwide gross.
2. Have a zero worldwide gross (*A zero worldwide gross is probably an incorrect value*).

# A Assignment templates and Datasets

Assignment templates and datasets used in the book can be found [here](#)