ITSB | Eco Industry Oriented University

sinarmas

DATA SCIENCE PROGRAMMING

Study Case Using R and Python

Writer:

Bakti Siregar, M.Sc., CDS.

D'SCIENCELABS
Smart Idea

Kampus Merdeka
INDONESIA JAYA

First Edition

Data Science Programming

Study Case Using R and Python

Bakti Siregar, M.Sc.,CDS

Table of contents

In today's digital age, data plays a crucial role in decision-making across various industries. Data Science Programming is essential for extracting meaningful insights from large datasets, automating analytical processes, and developing predictive models. Proficiency in data science programming enables professionals to analyze data effectively, optimize workflows, and create intelligent systems that drive innovation and technological advancement.

This module provides a comprehensive introduction to the fundamental concepts and practical applications of programming in data science. It covers key topics such as data manipulation, statistical analysis, machine learning, and automation using widely recognized programming languages like Python and R. Readers will have the opportunity to gain hands-on experience in handling real-world data, creating insightful visualizations, and applying statistical models to uncover patterns and trends.

Furthermore, the module explores data preprocessing, transformation, and integration, which are essential steps in preparing raw data for analysis. Readers will also develop practical skills in debugging, testing, and optimizing code to enhance efficiency and accuracy in data-driven projects.

Preface

About the Writer

Bakti Siregar, M.Sc., CDS works as a Lecturer at the ITSB Data Science Program. He earned his Master's degree from the Department of Applied Mathematics at National Sun Yat Sen University, Taiwan. In addition to teaching, Bakti also works as a Freelance Data Scientist for leading companies such as JNE, Samora Group, Pertamina, and PT. Green City Traffic.

He has a strong enthusiasm for projects (and teaching) in the fields of Big Data Analytics, Machine Learning, Optimization, and Time Series Analysis, particularly in finance and investment. His core expertise lies in statistical programming languages such as R Studio and Python. He is also experienced in implementing database systems like MySQL/NoSQL for data management and is proficient in using Big Data tools such as Spark and Hadoop.

Some of his projects can be viewed here: Rpubs, Github, Website, and Kaggle

---

Acknowledgments

Data Science Programming is a crucial skill in analyzing and processing large-scale data. This module covers essential programming techniques for data science, including:

- A solid foundation in **Python and R programming**
- The ability to **manipulate, analyze, and visualize data** effectively
- A deep understanding of **how domain knowledge enhances data analysis**
- Practical skills to apply **Data Science techniques in real-world scenarios**

This book are designed for beginners who aim to build a strong foundation in **Data Science Programming** while appreciating the critical role of **domain expertise**.

I appreciate the active participation of learners, whose questions and discussions enriched the training experience. I hope this material serves as a practical guide for applying programming in data science projects.

Feedback & Suggestions

Your feedback is essential in improving this module. We invite all participants to share their thoughts on the content, structure, and clarity of the materials. Suggestions for additional topics or areas requiring further explanation are highly appreciated.

With your support and contributions, we aim to refine this E-book, making it a more comprehensive resource for **Data Science Programming**. Thank you for your participation!

For feedback and suggestions, feel free to contact:

- dsciencelabs@outlook.com
- siregarbakti@gmail.com
- siregarbakti@itsb.ac.id

Introduction

In today's digital age, **data** is one of the most valuable assets for businesses, governments, and researchers. **Data Science** is the key to unlocking insights from vast amounts of information, driving innovation, and enhancing decision-making.

What is Data Science?

Data Science is an interdisciplinary field that combines statistics, mathematics, computer science, and domain knowledge to extract insights and meaningful patterns from structured and unstructured data. It involves techniques such as data analysis, machine learning, artificial intelligence (AI), and big data processing to support decision-making and automation.

Data Science is an interdisciplinary field that integrates:

- **Statistics and Mathematics** for data analysis
- **Programming (Python/R)** for data manipulation and automation
- **Machine Learning and AI** for predictive analytics and automation
- **Domain Knowledge** to provide meaningful context and interpretation

The Role of Domain Knowledge

While technical skills such as coding and statistical modeling are essential, **domain knowledge** is equally crucial. It helps in:

- Identifying relevant data sources and features for analysis
- Understanding the real-world implications of data-driven insights
- Making informed business or research decisions based on data

Key Components of Data Science

Data Science is a multi-step process that transforms raw data into valuable insights and actionable solutions. To achieve this, several key components work together, ensuring that data is collected, processed, analyzed, and utilized effectively. Below are the core components of Data Science:

- **Data Collection :** Gathering data from various sources (databases, APIs, sensors, web scraping).
- **Data Cleaning & Preprocessing :** Removing noise, handling missing values, and transforming data for analysis.
- **Exploratory Data Analysis (EDA) :** Identifying trends, patterns, and relationships in the data.
- **Machine Learning & AI :** Building predictive models and automating decision-making.
- **Data Visualization :** Communicating insights using charts, graphs, and dashboards.
- **Big Data Processing :** Managing large datasets using distributed computing (Hadoop, Spark).
- **Deployment & Decision-Making :** Implementing models into real-world applications.

Data Science Project Ideas

Untuk memahami dan menguasai konsep-konsep dalam Data Science, mengerjakan proyek praktis adalah cara terbaik. Dalam dokumen ini, kita akan membahas beberapa ide proyek Data Science yang dikategorikan berdasarkan tingkat kesulitan, mulai dari pemula hingga tingkat lanjut.

Here are some project list topics for "Applied of Data Science Across Industries":

#	Project Name	Description	Tools/Techniques
Beginner-Level Projects			
1	Exploratory Data Analysis (EDA) on Titanic Dataset	Analyze survival rates based on passenger data.	`pandas`, `ggplot2`, `dplyr`
2	Movie Recommendation System	Build a recommendation system using collaborative filtering.	MovieLens, `recommenderlab`
3	Stock Market Analysis	Perform trend analysis on historical stock data.	`ggplot2`, `matplotlib`, `pandas`
4	Sentiment Analysis on Twitter Data	Classify tweets as positive, negative, or neutral using NLP.	`rtweet`, `tidytext`
5	Fake News Detection	Develop a Machine Learning model to differentiate fake news.	`TF-IDF`, `Naïve Bayes`
Intermediate-Level Projects			

#	Project Name	Description	Tools/Techniques
6	Customer Segmentation using Clustering	Group customers based on shopping behavior using K-Means or DBSCAN.	`cluster`, `sklearn`
7	Churn Prediction Model	Predict whether a customer will stop using a service.	`Random Forest`, `XGBoost`
8	Time Series Forecasting on Sales Data	Predict future sales using time-series models.	`ARIMA`, `Prophet`, `LSTM`
9	Credit Card Fraud Detection	Build a classification model to detect fraudulent transactions.	`scikit-learn`, `XGBoost`
10	Chatbot using NLP	Create an AI chatbot for conversations.	`spaCy`, `NLTK`, `BERT`
Advanced-Level Projects			
11	Autonomous Vehicle Lane Detection	Use Computer Vision and OpenCV to detect road lanes.	`OpenCV`, `Deep Learning`
12	AI-Powered Resume Parser	Build an NLP model to extract key information from resumes.	`spaCy`, `TensorFlow`
13	Image Caption Generator	Generate automatic image descriptions using CNN + LSTM.	`TensorFlow`, `PyTorch`
14	Speech Emotion Recognition	Analyze voice data and classify emotions.	`Librosa`, `Deep Learning`
15	Medical Diagnosis with Deep Learning	Detect diseases in medical images (e.g., X-rays).	`CNN`, `Keras`, `PyTorch`
16	Real-Time Object Detection using YOLO	Develop an object detection system for real-time applications.	`YOLO`, `OpenCV`, `Deep Learning`

Why Learn Data Science Programming?

Programming is the foundation of Data Science, enabling professionals to:

- Process and clean raw data efficiently

- Perform **exploratory data analysis (EDA)** to uncover trends and patterns
- Build **machine learning models** to make accurate predictions
- Create compelling **data visualizations** for effective storytelling

The two most widely used programming languages in Data Science are **Python and R**, each with distinct advantages.

Python vs R for Data Science

Python and R are the two most popular programming languages for **Data Science and Analytics**. Both have **strong libraries, statistical tools, and machine learning capabilities**, but they excel in different areas. The following video is a detailed comparison to help determine the best choice based on use cases.

Aspect	Python	R
Ease of Learning	Intuitive, beginner-friendly	More specialized, statistical focus
Primary Use Cases	AI, Machine Learning, Web Development	Statistical computing, data visualization
Key Libraries	`pandas`, `numpy`, `matplotlib`, `seaborn`, `plotly`,`scikit-learn`	`tidyverse`, `dplyr`, `ggplot2`, `plotly`, `caret`
Performance	Optimized for large-scale computation	Designed for in-depth statistical analysis
Community Support	Industry-wide adoption	Preferred in academia & research

Key Takeaways:

- **Python** is ideal for AI, Machine Learning, and large-scale data analysis.
- **R** excels in statistical analysis and data visualization.
- **Both rogramming languages** have their unique strengths, and mastering both can be highly beneficial.

Part I

Programming

Chapter 1

Basic Programming

1.1 Data Types and Basic Operations

Every programming language has **different data types** that define the nature of stored values. The fundamental types in **Python and R** include:

Data Type	Python Example	R Example	Description
Integer (int)	`x = 10`	`x <- 10`	Whole numbers
Float (double in R)	`y = 3.14`	`y <- 3.14`	Decimal numbers
String (chr in R)	`name = "Alice"`	`name <- "Alice"`	Text data
Boolean (logical in R)	`is_valid = True`	`is_valid <- TRUE`	True or False
List (Vector in R)	`numbers = [1, 2, 3]`	`numbers <- c(1, 2, 3)`	Ordered collection of values
Dictionary (Named List in R)	`person = {"name": "Bob", "age": 25}`	`person <- list(name="Bob", age=25)`	Key-value storage

1.2 Variable Declaration

In both Python and R, variables do not need explicit declaration.

1.2.1 Python Code

```
x = 10              # Integer
y = 3.14            # Float
name = "Alice"      # String
is_valid = True     # Boolean
```

1.2.2 R Code

```r
x <- 10            # Integer
y <- 3.14          # Numeric
name <- "Alice"    # Character
is_valid <- TRUE   # Logical
```

1.3 Basic Data Manipulation

Operations on variables such as reassignment, updating values, and basic computations.

1.3.1 Python Code

```python
x = x + 5                       # Updating value
name = name + " Johnson"        # String concatenation
print(name)                     # Output: Alice Johnson
```

1.3.2 R Code

```r
x <- x + 5                      # Updating value
name <- paste(name, "Johnson")  # String concatenation
print(name)                     # Output: Alice Johnson
```

1.4 Basic Arithmetic Operations

Arithmetic operations are fundamental for numerical computations.

Operation	Python Example	R Example	Description
Addition (+)	`a + b`	`a + b`	Adds two numbers
Subtraction (-)	`a - b`	`a - b`	Subtracts one number from another
Multiplication (*)	`a * b`	`a * b`	Multiplies two numbers
Division (/)	`a / b`	`a / b`	Divides one number by another
Exponentiation (**) / (^ in R)	`a ** 2`	`a ^ 2`	Raises a number to a power
Modulo (%)	`a % b`	`a %% b`	Returns remainder of division

1.4.1 Python Code

```python
a = 10
b = 3
print(a + b)   # Addition
```

```
print(a - b)   # Subtraction
print(a * b)   # Multiplication
print(a / b)   # Division
print(a ** 2)  # Exponentiation
print(a % b)   # Modulo
```

1.4.2 R Code

```
a <- 10
b <- 3
print(a + b)   # Addition
print(a - b)   # Subtraction
print(a * b)   # Multiplication
print(a / b)   # Division
print(a ^ 2)   # Exponentiation
print(a %% b)  # Modulo
```

1.5 String Operations

String operations are used to manipulate text data.

Operation	Python Example	R Example	Description
Uppercase	`text.upper()`	`toupper(text)`	Convert to uppercase
Lowercase	`text.lower()`	`tolower(text)`	Convert to lowercase
Length	`len(text)`	`nchar(text)`	Get string length
Substring	`text[0:5]`	`substr(text, 1, 5)`	Extract part of string

1.5.1 Python Code

```
text = "Hello, World!"   # Variable declaration
print(text.upper())      # Convert to uppercase
print(text.lower())      # Convert to lowercase
print(len(text))         # Get string length
print(text[0:5])         # Slicing (first 5 characters)
```

1.5.2 R Code

```r
text <- "Hello, World!"   # variable declaration
print(toupper(text))      # Convert to uppercase
print(tolower(text))      # Convert to lowercase
print(nchar(text))        # Get string length
print(substr(text, 1, 5)) # Extract first 5 characters
```

1.6 Logical and Comparison Operators

Logical and comparison operators help in decision-making.

Operator	Description	Python Example	R Example
Equal (==)	Checks if values are equal	`x == y`	`x == y`
Not Equal (!=)	Checks if values are different	`x != y`	`x != y`
Greater (>)	Checks if left value is greater	`x > y`	`x > y`
Less (<)	Checks if left value is smaller	`x < y`	`x < y`
AND (and in Python, & in R)	Both conditions must be true	`(x > 5) and (y < 10)`	`(x > 5) & (y < 10)`
OR (or in Python, | in R)	At least one condition is true	`(x > 5) or (y > 10)`	`(x > 5) | (y > 10)`

1.6.1 Python Code

```python
x = 10
y = 5
print(x == y)                # False
print(x != y)                # True
print(x > y)                 # True
print(x < y)                 # False
print((x > 5) and (y < 10))  # True
print((x > 5) or (y > 10))   # True
```

1.6.2 R Code

```r
x <- 10
y <- 5
```

```
print(x == y)               # FALSE
print(x != y)               # TRUE
print(x > y)                # TRUE
print(x < y)                # FALSE
print((x > 5) & (y < 10))   # TRUE
print((x > 5) | (y > 10))   # TRUE
```

1.7 Data Type Conversion

Data type conversion allows changing one data type to another.

Conversion	Python Example	R Example	Description
String to Integer	`int("100")`	`as.numeric("100")`	Convert text to number
Integer to String	`str(100)`	`as.character(100)`	Convert number to text
Float to Integer	`int(10.5)`	`as.integer(10.5)`	Convert decimal to whole number

1.7.1 Python Code

```
num_str = "100"
num = int(num_str)              # Convert string to integer
print(num + 10)                 # Output: 110
```

```
num_float = 10.5
num_int = int(num_float)        # Convert float to integer
print(num_int)                  # Output: 10
```

1.7.2 R Code

```
num_str <- "100"
num <- as.numeric(num_str)      # Convert string to number
print(num + 10)                 # Output: 110

num_float <- 10.5
num_int <- as.integer(num_float) # Convert float to integer
print(num_int)                  # Output: 10
```

1.8 Practicum

1.8.1 Identifying Data Types

Determine the data types of the following variables in Python and R:

```
# R
 a = 42
 b = 3.14
 c = "Hello"
 d = FALSE
 e = [1, 2, 3]
 f = {"name": "Alice", "age": 25}
```

Questions:

1. Identify the data type of each variable above.
2. Print the data type of each variable using `type()` (Python) and `class()` (R).

1.8.2 Variables and Data Manipulation

Create the following variables in **Python** and **R**:

```
# R
 x = 20
 y = 5
 text1 = "Data"
 text2 = "Science"
```

Questions:

1. Update the value of `x` by adding `10`.
2. Concatenate `text1` and `text2` into `"Data Science"`.
3. Convert the concatenated text to uppercase.

1.8.3 Arithmetic Operations

Given the following variables:

```
# R
 a = 15
 b = 4
```

Questions:

1. Compute the sum, difference, product, division, and modulo of `a` and `b`.
2. Compute `a` raised to the power of `b`.
3. Create a new variable `c = a / b` and convert it to an **integer**.

1.8.4 String Operations

Given the following text:

```
# R
 text = "Hello, Data Science!"
```

Questions:

1. Extract the **first 5 characters** from the text.
2. Count the number of characters in the text.
3. Convert the text to lowercase.

1.8.5 Comparison and Logical Operators

Given the following variables:

```
# R
 x = 7
 y = 10
```

Questions:

1. Check if `x` is greater than `y`.
2. Check if `x` is less than or equal to `y`.
3. Check if `x` is **not equal to** `y`.
4. Evaluate the expression `(x > 5) AND (y < 20)`.

1.8.6 Data Type Conversion

Given the following variables:

```
# R
 num_str = "123"
 num_float = 45.67
```

Questions:

1. Convert `num_str` to an integer and add `10`.
2. Convert `num_float` to an integer.
3. Convert the converted `num_float` back to a string.

1.9 Bonus Challenge

Create an interactive program that asks the user to input:

1. Name
2. Age
3. Hometown

Then, print the output as follows:

```
"Hello [Name], you are [Age] years old and from [Hometown]."
```

Happy coding! If you need any help, feel free to ask **Chat-GPT**.

Chapter 2

Syntax and Control Flow

Control flow statements determine the order in which code is executed, allowing programs to make logical decisions, perform repetitive tasks, and handle errors efficiently. To learn more, consider watching the following video:

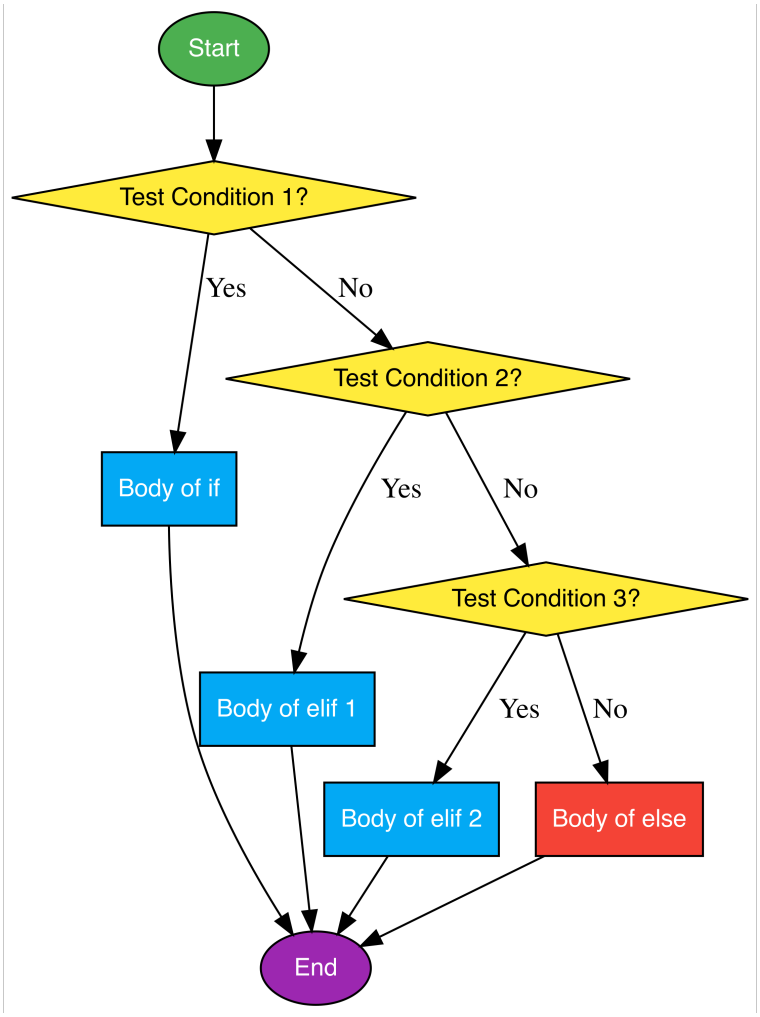
There are three main types of control flow statements:

- Conditional Statements → Enable decision-making in a program (e.g., if, if-else, if-elif-else).
- Loops → Facilitate repetition of actions (e.g., for, while, with control keywords like break and continue).
- Error Handling → Ensure smooth program execution by managing unexpected errors (e.g., try-except in Python, tryCatch in R).

2.1 Conditional Statements

Conditional statements let a program **execute different code** depending on whether a condition is **true or false**.

- **`if` statement** → Executes code only if a condition is **true**.
- **`if-else` statement** → Executes one block if **true**, another if **false**.
- **`if-elif-else` statement** → Checks multiple conditions.



2.1.1 Simple Example

Check if a number is positive, negative, or zero.

Python Code

```
x = 10

if x > 0:
    print("Positive number")
elif x == 0:
    print("Zero")
else:
    print("Negative number")
```

```
Positive number
```

R Code

```r
x <- 10

if (x > 0) {
  print("Positive number")
} else if (x == 0) {
  print("Zero")
} else {
  print("Negative number")
}
```

```
[1] "Positive number"
```

2.1.2 Medium Example

Check if a number is even or odd, with an additional condition.

Python Code

```python
try:
    x = int(input("Enter a number: "))

    if x % 2 == 0:
        print(f"{x} is even.")
        if x % 4 == 0:
            print(f"{x} is also a multiple of 4.")
    else:
        print(f"{x} is odd.")

except ValueError:
    print("Invalid input! Please enter a valid integer.")
```

R Code

```r
x <- as.integer(readline("Enter a number: "))

if (x %% 2 == 0) {
  print(paste(x, "is even."))
  if (x %% 4 == 0) {
    print(paste(x, "is also a multiple of 4."))
  }
} else {
  print(paste(x, "is odd."))
}
```

2.1.3 Complex Example

Categorizing a person's age group.

Python Code

```python
age = int(input("Enter your age: "))

if age < 0:
    print("Invalid age")
elif age <= 12:
    print("You are a child.")
elif age <= 19:
    print("You are a teenager.")
elif age <= 59:
    print("You are an adult.")
else:
    print("You are a senior.")
```

R Code

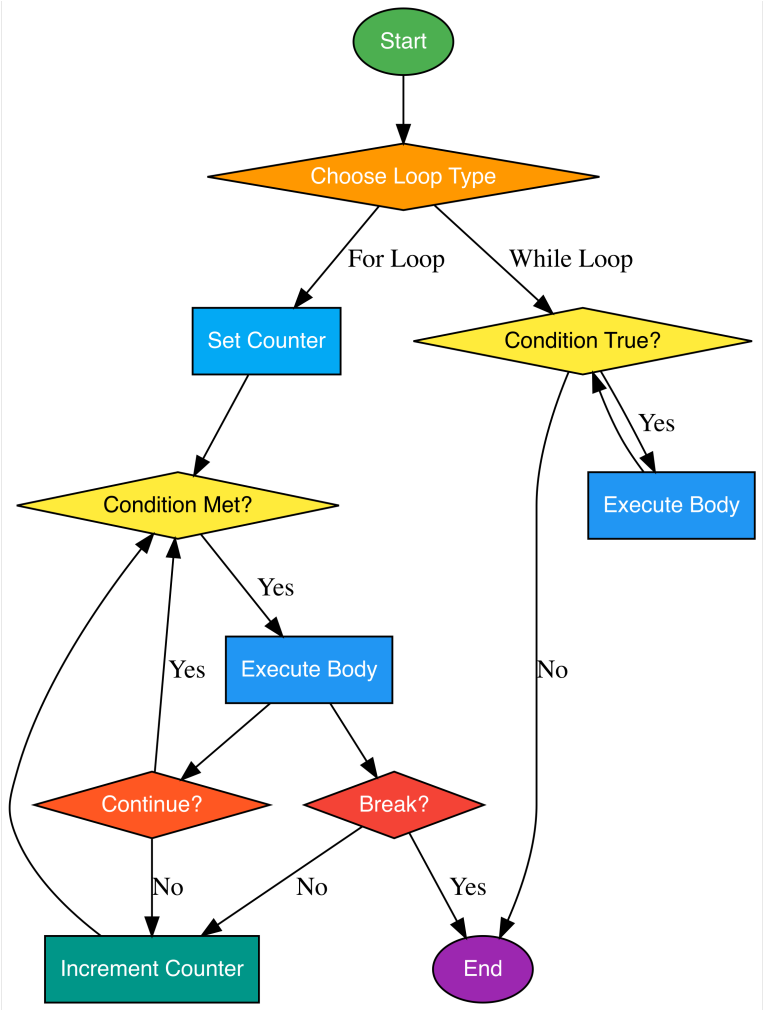
```r
age <- as.integer(readline("Enter your age: "))

if (age < 0) {
  print("Invalid age")
} else if (age <= 12) {
  print("You are a child.")
} else if (age <= 19) {
  print("You are a teenager.")
} else if (age <= 59) {
  print("You are an adult.")
} else {
  print("You are a senior.")
}
```

2.2 Loops

Loops allow programs to repeat actions multiple times.

- For Loop → Used when the number of iterations is known. The counter is set, the condition is checked, the code executes, then the counter increments until the condition is no longer met.
- While Loop → Used when looping should continue as long as the condition is true. If the condition remains valid, the code executes and is checked again until the condition becomes false.
- Break → Stops the loop early, immediately exiting the loop without completing all iterations.
- Continue → Skips the current iteration without stopping the loop, returning directly to the condition check for the next iteration. The loop stops when the condition is no longer met or a break statement is used. The loop continues if the condition remains true unless a break occurs.



2.2.1 Simple Example

Print numbers from 1 to 5 using a for loop.

Python Code

```
for i in range(1, 6):
    print(i)
```

```
1
2
3
4
5
```

R Code

```r
for (i in 1:5) {
  print(i)
}
```

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

2.2.2 Medium Example

Using a while loop to print numbers up to a limit.

Python Code

```python
x = 1
while x <= 5:
    print(x)
    x += 1
```

```
1
2
3
4
5
```

R Code

```r
x <- 1
while (x <= 5) {
  print(x)
  x <- x + 1
}
```

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

2.2.3 Complex Example

Using break and continue to modify loop behavior.

Python Code

```python
for i in range(1, 11):
    if i == 5:
        continue  # Skip number 5
    if i == 8:
        break     # Stop when i = 8
    print(i)
```

```
1
2
3
4
6
7
```

R Code

```r
for (i in 1:10) {
  if (i == 5) {
    next          # Skip number 5
  }
  if (i == 8) {
    break         # Stop when i = 8
  }
  print(i)
}
```

```
[1] 1
[1] 2
[1] 3
[1] 4
[1] 6
[1] 7
```

2.3 Error Handling

Error handling allows a program to recover from unexpected issues rather than terminating abruptly.

- try-except (Python) → Captures errors and ensures the program continues running.
- tryCatch (R) → Provides similar functionality in R, allowing controlled error management.

2.3.1 Simple Example

Handling incorrect date formats during user input

Python Code

```python
from datetime import datetime

try:
    date_str = input("Enter date (YYYY-MM-DD): ")  # Input may have wrong format
    date_obj = datetime.strptime(date_str, "%Y-%m-%d")
    print(f"Entered date: {date_obj}")
except ValueError:
    print("Error: Date format must be YYYY-MM-DD!")
```

R Code

```r
safe_date <- function() {
  date_str <- readline("Enter date (YYYY-MM-DD): ")
  tryCatch({
    date_obj <- as.Date(date_str, format="%Y-%m-%d")
    if (is.na(date_obj)) stop("Incorrect date format!")
    print(paste("Entered date:", date_obj))
  }, error = function(e) {
    print(paste("Error:", e$message))
  })
}

safe_date()
```

```
Enter date (YYYY-MM-DD):
[1] "Error: Incorrect date format!"
```

2.3.2 Medium Example

Handling missing values and invalid dates in datasets.

Python Code

```python
# inport library
import pandas as pd

# dataset example (this is dictionary type)
data = {"event": ["A", "B", "C"], "date": ["2024-01-15", "2024-15-10", None]}
df = pd.DataFrame(data)

# error handaling program
try:
    df["date"] = pd.to_datetime(df["date"], errors="coerce")  # dates to NaT
    df.dropna(subset=["date"], inplace=True)  # Remove rows with invalid dates
    print(df)
except Exception as e:
    print(f"Error processing dates: {e}")
```

R Code

```r
library(dplyr)
library(lubridate)

# dataset example (this is dictionary type)
data <- data.frame(event = c("A", "B", "C"),
                   date = c("2024-01-15", "2024-15-10", NA))

# error handaling program
safe_process_dates <- function(df) {
  tryCatch({
    df <- df %>%
      mutate(date = ymd(date)) %>%  # Convert dates
      filter(!is.na(date))          # Remove invalid dates

    print(df)
  }, error = function(e) {
    print(paste("Error processing dates:", e$message))
  })
}

safe_process_dates(data)
```

2.3.3 Complex Example

This Python code replicates the R functionality while improving date handling.

Python Code

```python
import pandas as pd
from dateutil import parser
import warnings

# Example dataset with various incorrect date formats
data = pd.DataFrame({
    "event": ["A", "B", "C", "D", "E"],
    "date": ["2024-01-15", "2024-15-10", None, "15-03-2024", "2024/04/05"]
})

def safe_process_dates(df):
    def parse_date(date_str):
        try:
            return parser.parse(date_str, dayfirst=True).strftime("%d-%m-%Y")
        except (ValueError, TypeError):
            return None

    # Apply date parsing
    df["parsed_date"] = df["date"].apply(parse_date)
```

```
    # Find invalid dates
    invalid_dates = df[df["parsed_date"].isna()]["date"].dropna().tolist()

    if invalid_dates:
        warnings.warn(f"Some dates are invalid: {', '.join(invalid_dates)}")

    # Filter out invalid dates
    df = df.dropna(subset=["parsed_date"])

    print(df)

safe_process_dates(data)
```

```
  event        date parsed_date
0     A  2024-01-15  15-01-2024
1     B  2024-15-10  15-10-2024
3     D  15-03-2024  15-03-2024
4     E  2024/04/05  04-05-2024
```

R Code

```
library(dplyr)
library(lubridate)

# Example dataset with various incorrect date formats
data <- data.frame(event = c("A", "B", "C", "D", "E"),
                   date = c("2024-01-15", "2024-15-10", NA,
                            "15-03-2024", "2024/04/05"))

# Function to handle errors in date conversion
safe_process_dates <- function(df) {
  tryCatch({
    df <- df %>%
      mutate(
        parsed_date = parse_date_time(date,
                                      orders = c("ymd", "dmy", "mdy", "Y/m/d"),
                                      quiet = TRUE)
      ) %>%
      filter(!is.na(parsed_date)) %>%   # Remove dates that failed to convert
      mutate(formatted_date = format(parsed_date, "%d-%m-%Y")) # Reformat dates

    # Check if there are any invalid dates
    invalid_dates <- df$date[is.na(df$parsed_date)]
    if (length(invalid_dates) > 0) {
      warning("Some dates are invalid: ", paste(invalid_dates, collapse = ", "))
    }

    print(df)
```

```
  }, error = function(e) {
    print(paste("Error processing dates:", e$message))
  })
}

safe_process_dates(data)
```

```
  event       date parsed_date formatted_date
1     A 2024-01-15  2024-01-15     15-01-2024
2     D 15-03-2024  2024-03-15     15-03-2024
3     E 2024/04/05  2024-04-05     05-04-2024
```

Explanation:

- `dateutil.parser.parse()` is used for flexible date recognition.
- Handles multiple date formats including `YYYY-MM-DD`, `DD-MM-YYYY`, `YYYY/MM/DD`, etc.
- Removes invalid dates and displays warnings for them.
- Formats valid dates to `YYYY-MM-DD` for consistency.

2.4 Best Practices

In this section you will learn the **Best Practices** about Syntax & Control Flow in Python and R:

2.4.1 Readability & Formatting

- **Follow PEP 8** for clean and readable code.
- **Use consistent indentation (4 spaces per level).**
- **Keep line length 79 characters.**
- **Use meaningful variable and function names.**

Python Code

```
#  Good: Readable and follows PEP 8
def calculate_total(price, tax_rate):
    "Calculate the total price after tax."
    total = price + (price * tax_rate)
    return total
```

```
#  Bad: Poor formatting and unclear naming
def calc(p, t): return p+(p*t) # One-liner (not recommended)
```

R Code

```
#  Good: Readable and follows conventions
total_price <- function(price, tax_rate) {
```

```r
  "Calculate the total price after tax."
  total <- price + (price * tax_rate)
  return(total)
}
```

```r
#  Bad: Poor formatting and unclear naming
total <- function(p, t) { p + (p * t) } # One-liner (not recommended)
```

2.4.2 Efficient Control Flow

- **Use `if-elif-else` correctly** to avoid redundant checks.
- **Avoid deep nesting**; use guard clauses to improve readability.

Python Code

```python
#  Good: Uses guard clause to return early
def check_access(user):
    if not user.is_authenticated:
        return "Access Denied"

    if user.is_admin:
        return "Access Granted: Admin"

    return "Access Granted: User"
```

```python
#  Bad: Deeply nested structure
def check_access(user):
    if user.is_authenticated:
        if user.is_admin:
            return "Access Granted: Admin"
        else:
            return "Access Granted: User"
    else:
        return "Access Denied"
```

R Code

```r
#  Good: Uses early return
check_access <- function(user) {
  if (!user$authenticated) {
    return("Access Denied")
  }

  if (user$admin) {
    return("Access Granted: Admin")
  }

  return("Access Granted: User")
```

```
}
```

2.4.3 Looping Best Practices

- **Use list comprehensions for simple loops.**
- **Use `enumerate()` instead of manually managing indexes.**
- **Use `zip()` for iterating over multiple lists simultaneously.**

Python Code

```python
#  Good: List comprehension for efficiency
squares = [x**2 for x in range(10) if x % 2 == 0]

#  Good: Using enumerate
names = ["Alice", "Bob", "Charlie"]
for index, name in enumerate(names, start=1):
    print(f"{index}: {name}")

#  Good: Using zip()
keys = ["name", "age", "city"]
values = ["Alice", 25, "New York"]
person = dict(zip(keys, values))
```

```r
#  Good: Vectorized operation
squares <- (0:9)^2

#  Good: Using sapply
names <- c("Alice", "Bob", "Charlie")
print(sapply(seq_along(names), function(i) paste(i, names[i])))
```

```
[1] "1 Alice"   "2 Bob"     "3 Charlie"
```

2.4.4 Common Mistakes in Loops

- **Don't modify lists while iterating** (use a copy instead).
- **Use `break` and `continue` sparingly** to maintain readability.

Python Code

```python
#  Good: Iterating over a copy when modifying a list
numbers = [1, 2, 3, 4, 5]
for num in numbers[:]:  # Copy of the list
    if num % 2 == 0:
        numbers.remove(num)
```

```python
#  Bad: Modifying a list while iterating (can cause unexpected behavior)
for num in numbers:
```

```
    if num % 2 == 0:
        numbers.remove(num)
```

R Code

```
#  Good: Using which to filter elements
numbers <- c(1, 2, 3, 4, 5)
numbers <- numbers[numbers %% 2 != 0]
```

2.4.5 Cleaner Conditionals

Use `match-case` instead of long `if-elif` chains when checking specific values **Python 3.10+**.

Python

```
#  Good: Using match-case
def get_status(code):
    match code:
        case 200:
            return "OK"
        case 404:
            return "Not Found"
        case 500:
            return "Internal Server Error"
        case _:
            return "Unknown Status"
```

```
#  Bad: Using multiple if-elif
def get_status(code):
    if code == 200:
        return "OK"
    elif code == 404:
        return "Not Found"
    elif code == 500:
        return "Internal Server Error"
    else:
        return "Unknown Status"
```

R Code

```
#  Good: Using switch
get_status <- function(code) {
  switch(as.character(code),
         "200" = "OK",
         "404" = "Not Found",
         "500" = "Internal Server Error",
         "Unknown Status")
}
```

2.4.6 Error Handling

- **Catch only specific exceptions** instead of using a general `except` clause.
- **Use `finally` for cleanup operations** (e.g., closing files, releasing resources).

Python Code

```python
#  Good: Handling specific exceptions
try:
    number = int(input("Enter a number: "))
    result = 10 / number
except ValueError:
    print("Invalid input! Please enter a number.")
except ZeroDivisionError:
    print("Cannot divide by zero.")
finally:
    print("Execution complete.")
```

R Code

```r
#  Good: Handling specific exceptions
safe_divide <- function(a, b) {
  tryCatch({
    result <- a / b
    return(result)
  }, warning = function(w) {
    message("Warning: ", w$message)
  }, error = function(e) {
    message("Error: Cannot divide by zero.")
  })
}
```

2.4.7 Resource Management

Use `with` statements when working with files to ensure proper cleanup.

Python Code

```python
#  Good: Using with statement (auto-closes file)
with open("data.txt", "r") as file:
    content = file.read()
```

```python
#  Bad: Forgetting to close the file
file = open("data.txt", "r")
content = file.read()
file.close()
```

R Code

```r
#  Good: Using on.exit() to clean up
read_data <- function(file) {
  con <- file(file, "r")
  on.exit(close(con))
  data <- readLines(con)
  return(data)
}
```

2.4.8 Mutable Default Arguments

Use immutable types (e.g., `None`) as default arguments instead of mutable ones.

Python Code

```python
#  Good: Using None to avoid unintended behavior
def add_item(item, items=None):
    if items is None:
        items = []
    items.append(item)
    return items
```

R Code

```r
#  Good: Using NULL to avoid unintended behavior
add_item <- function(item, items = NULL) {
  if (is.null(items)) {
    items <- list()
  }
  items <- append(items, item)
  return(items)
}
```

2.4.9 Using Generators for

Use `yield` for efficient memory usage when handling large datasets.

Python Code

```python
#  Good: Using generator function
def count_up_to(n):
    count = 1
    while count <= n:
        yield count
        count += 1

for num in count_up_to(5):
    print(num)
```

R Code

```r
#  Good: Using closures to generate a sequence
count_up_to <- function(n) {
  count <- 1
  function() {
    if (count <= n) {
      result <- count
      count <<- count + 1
      return(result)
    } else {
      return(NULL)
    }
  }
}

counter <- count_up_to(5)
while (!is.null(value <- counter())) {
  print(value)
}
```

By following these best practices, you can write **clean, efficient, and maintainable** Python and R code.

2.5 Practicum

Independent Practice: Conditional Statements and Loops in Python & R

2.5.1 Objective

1. Understand and implement **conditional statements** (`if`, `if-else`, `if-elif-else`).
2. Apply **loops** (`for loop`, `while loop`, `break`, `continue`) to analyze a dataset.

Use the following **dummy dataset**:

ID	Name	Age	Salary	Position	Performance
1	Bagas	25	5000	Staff	Good
2	Joan	30	7000	Supervisor	Very Good
3	Alya	27	6500	Staff	Average
4	Dwi	35	10000	Manager	Good
5	Nabil	40	12000	Director	Very Good

2.5.2 Conditional Statements

Determine **bonus levels** based on employee **performance**:

- **Very Good** → 20% of salary
- **Good** → 10% of salary

- **Average** → 5% of salary

Your Task:

- Write a program in **Python and R** to calculate each employee's bonus.
- Display the output in this format:
  `"Name: Bagas, Bonus: 500"`

2.5.3 Loops (For & While)

1. Use a **for loop** to list employees with a salary greater than **6000**.

Expected Output:

```
Name: Joan, Salary: 7000
Name: Alya, Salary: 6500
Name: Dwi, Salary: 10000
Name: Nabil, Salary: 12000
```

2. Use a **while loop** to display employees until a **"Manager"** is found.

Expected Output:

```
Name: Bagas, Position: Staff
Name: Joan, Position: Supervisor
Name: Alya, Position: Staff
Name: Dwi, Position: Manager (Stop here)
```

3. Use **break** to stop the loop when an employee with a salary above **10,000** is found.

Expected Output:

```
Name: Bagas, Salary: 5000
Name: Joan, Salary: 7000
Name: Alya, Salary: 6500
Name: Dwi, Salary: 10000
(Stopped because Nabil has a salary above 10,000)
```

4. Use **continue** to **skip** employees with **"Average"** performance.

Expected Output:

```
Name: Bagas, Performance: Good
Name: Joan, Performance: Very Good
Name: Dwi, Performance: Good
Name: Nabil, Performance: Very Good
(Alya is skipped because the performance is "Average")
```

Submission Guidelines:

1. Submit your **Python and R** code using your `Google Colab` and `Rpubs`.
2. Ensure the output is displayed correctly.
3. Add comments in the code to explain your logic.

Chapter 3

Functions and Loops

3.1 Introduction

In programming, we often perform the same tasks repeatedly. **Functions** and **Loops** help us write cleaner, shorter, and more efficient code.

- **Function** is a block of code that can be called anytime to perform a specific task.
- **Loop** is used to run the same code repeatedly without rewriting it.

3.2 What Is a Function?

A function is a block of code designed to perform a specific task. Using functions helps us avoid redundant code.

3.2.1 Adding Two Numbers

This function takes two numbers as inputs and returns their sum.

Python Code

```python
# Function to add two numbers
def add_numbers(a, b):
    return a + b

print(add_numbers(5, 3))  # Output: 8
```

```
8
```

R Code

```r
# Function to add two numbers
add_numbers <- function(a, b) {
  return(a + b)
```

```
}

print(add_numbers(5, 3))  # Output: 8
```

```
[1] 8
```

3.2.2 Rectangle Properties

This function calculates the area and perimeter of a rectangle, given its length and width.

Python Code

```
# Function to calculate area and perimeter of a rectangle
def rectangle_properties(length, width):
    area = length * width
    perimeter = 2 * (length + width)
    return {"area": area, "perimeter": perimeter}

print(rectangle_properties(5, 3))
```

```
{'area': 15, 'perimeter': 16}
```

```
# Output: {'area': 15, 'perimeter': 16}
```

3.2.2.1 R Code

```
# Function to calculate area and perimeter of a rectangle
rectangle_properties <- function(length, width) {
  area <- length * width
  perimeter <- 2 * (length + width)
  return(list(area = area, perimeter = perimeter))
}

print(rectangle_properties(5, 3))
```

```
$area
[1] 15

$perimeter
[1] 16
```

```
# Output: $area [1] 15, $perimeter [1] 16
```

3.2.3 Comparing Two Datasets

This function analyzes two datasets by calculating their mean, median, and standard deviation, useful in data analysis.

```python
import statistics
from tabulate import tabulate

# Function to compare two datasets
def compare_data(group1, group2):
    return {
        "group1": {
            "mean": statistics.mean(group1),
            "median": statistics.median(group1),
            "std_dev": statistics.stdev(group1)
        },
        "group2": {
            "mean": statistics.mean(group2),
            "median": statistics.median(group2),
            "std_dev": statistics.stdev(group2)
        }
    }

# Sample datasets
data1 = [10, 20, 30, 40, 50]
data2 = [15, 25, 35, 45, 55]

# Get results
results = compare_data(data1, data2)

# Convert results to a table format
table = [
    ["Metric", "Group 1", "Group 2"],
    ["Mean", results["group1"]["mean"], results["group2"]["mean"]],
    ["Median", results["group1"]["median"], results["group2"]["median"]],
    ["Standard Deviation", results["group1"]["std_dev"], results["group2"]["std_dev"]]
]

# Print table
print(tabulate(table, headers="firstrow", tablefmt="grid"))
```

```
+--------------------+-----------+-----------+
| Metric             |   Group 1 |   Group 2 |
+====================+===========+===========+
| Mean               |   30      |   35      |
+--------------------+-----------+-----------+
| Median             |   30      |   35      |
+--------------------+-----------+-----------+
| Standard Deviation |   15.8114 |   15.8114 |
+--------------------+-----------+-----------+
```

R Code

```r
# Load library
library(knitr)

# Function to compare two datasets
compare_data <- function(group1, group2) {
  data.frame(
    Statistic = c("Mean", "Median", "Std Dev"),
    Group1 = round(c(mean(group1), median(group1), sd(group1)), 2),
    Group2 = round(c(mean(group2), median(group2), sd(group2)), 2)
  )
}

# Sample data
data1 <- c(10, 20, 30, 40, 50)
data2 <- c(15, 25, 35, 45, 55)

# Print as formatted table
kable(compare_data(data1, data2))
```

Statistic	Group1	Group2
Mean	30.00	35.00
Median	30.00	35.00
Std Dev	15.81	15.81

Functions save time by allowing code reuse, improve program organization and readability, and make debugging and future development easier.

3.3 What Is a Loop?

Loops allow us to execute the same code multiple times without rewriting it. Loops allow us to perform repetitive calculations for mathematical analysis and data processing. Types of Loops:

- **For Loop** – Used when the number of repetitions is known.
- **While Loop** – Used when repetitions depend on a condition.

3.3.1 Fibonacci Sequence

The Fibonacci sequence is a series of numbers where each number is the sum of the two preceding ones:

$$F(n) = F(n-1) + F(n-2)$$

Example: `$0,1,1,2,3,5,8,13,21,\dots$`

Python Code

```
def fibonacci(n):
    fib_series = [0, 1]
    for i in range(2, n):
        fib_series.append(fib_series[-1] + fib_series[-2])
    return fib_series

print(fibonacci(10))  # Output: [0, 1, 1, 2, 3, 5, 8, 13, 21, 34]
```

```
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34]
```

3.3.1.1 R Code

```
fibonacci <- function(n) {
  fib_series <- c(0, 1)
  for (i in 3:n) {
    fib_series <- c(fib_series, fib_series[i-1] + fib_series[i-2])
  }
  return(fib_series)
}

print(fibonacci(10))  # Output: 0 1 1 2 3 5 8 13 21 34
```

```
 [1]  0  1  1  2  3  5  8 13 21 34
```

3.3.2 Standard Deviation

Standard deviation measures how spread out the data is in a distribution:

$$\sigma = \sqrt{\frac{1}{n}\sum(x_i - \bar{x})^2}$$

Python Code

```
def standard_deviation(data):
    mean = sum(data) / len(data)
    variance = sum((x - mean) ** 2 for x in data) / len(data)
    return variance ** 0.5

data = [10, 20, 30, 40, 50]

print(f"Standard Deviation: {standard_deviation(data):.2f}")
```

```
Standard Deviation: 14.14
# Output: 14.14
```

R Code

```r
standard_deviation <- function(data) {
  mean_value <- mean(data)
  variance <- sum((data - mean_value) ^ 2) / length(data)
  return(sqrt(variance))
}

data <- c(10, 20, 30, 40, 50)

print(paste("Standard Deviation:", round(standard_deviation(data), 2)))
```

```
[1] "Standard Deviation: 14.14"
```

```r
# Output: 14.14
```

3.3.3 Simple Linear Regression

Linear regression is used to find the relationship between an independent variable X and a dependent variable Y:

$$Y = aX + b$$

where:

- a is the slope
- b is the intercept

Python Code

```python
import numpy as np

# Data (X: study hours, Y: exam scores)
X = np.array([1, 2, 3, 4, 5])
Y = np.array([2, 4, 5, 4, 5])

# Calculate slope (a) and intercept (b)
n = len(X)
sum_x, sum_y = sum(X), sum(Y)
sum_xy = sum(X * Y)
sum_x2 = sum(X ** 2)

a = (n * sum_xy - sum_x * sum_y) / (n * sum_x2 - sum_x ** 2)
b = (sum_y - a * sum_x) / n

print(f"Linear Regression: Y = {a:.2f}X + {b:.2f}")
```

```
Linear Regression: Y = 0.60X + 2.20
```

R Code

```r
# Data
X <- c(1, 2, 3, 4, 5)
Y <- c(2, 4, 5, 4, 5)

# Calculate slope (a) and intercept (b)
n <- length(X)
sum_x <- sum(X)
sum_y <- sum(Y)
sum_xy <- sum(X * Y)
sum_x2 <- sum(X^2)

a <- (n * sum_xy - sum_x * sum_y) / (n * sum_x2 - sum_x^2)
b <- (sum_y - a * sum_x) / n

print(paste("Linear Regression: Y =", round(a, 2), "X +", round(b, 2)))
```

```
[1] "Linear Regression: Y = 0.6 X + 2.2"
```

Functions and loops help us create simpler and more efficient code. By understanding these two concepts, we can write better and more readable programs.

3.4 Applied of Functions and Loops

Let's apply these **Functions and Loops** to real-world data science tasks, such as:

- Creating a dataset dynamically using functions and loops.
- Categorizing employees based on salary.
- Computing aggregated statistics (average salary, experience, etc.).
- Finding top-paid employees per job position.

3.4.1 Creating a Dataset

Python Code

```python
import pandas as pd
import random

def create_employee_dataset(num_employees):
    positions = {
        "Staff": (3000, 5000, 1, 5),
        "Supervisor": (5000, 8000, 5, 10),
        "Manager": (8000, 12000, 10, 15),
        "Director": (12000, 15000, 15, 25)
    }

    data = {
        "ID_Number": [],
        "Position": [],
```

```
        "Salary": [],
        "Age": [],
        "Experience": []
    }

    for _ in range(num_employees):
        id_number = random.randint(10000, 99999)
        position = random.choice(list(positions.keys()))
        salary = random.randint(positions[position][0], positions[position][1])
        experience = random.randint(positions[position][2], positions[position][3])
        age = experience + random.randint(22, 35)  # Ensuring age aligns with experience

        data["ID_Number"].append(id_number)
        data["Position"].append(position)
        data["Salary"].append(salary)
        data["Age"].append(age)
        data["Experience"].append(experience)

    return pd.DataFrame(data)

df = create_employee_dataset(20)
print(df)
```

```
    ID_Number    Position  Salary  Age  Experience
0       81265    Director   14129   43          15
1       70156    Director   13739   47          16
2       51249    Director   14322   49          22
3       74338     Manager    9443   34          12
4       20685       Staff    3682   35           1
5       76138  Supervisor    6469   31           5
6       10658    Director   13009   55          23
7       71600    Director   13287   51          19
8       38298    Director   13106   55          22
9       20101  Supervisor    5788   42           7
10      25628    Director   13910   49          21
11      43457       Staff    3955   30           5
12      20423    Director   14712   57          23
13      76416  Supervisor    6719   35           9
14      24459     Manager    9773   44          12
15      42241     Manager   10318   34          12
16      37494       Staff    3291   30           3
17      18085  Supervisor    5631   39           8
18      53829    Director   12036   50          25
19      84677    Director   13286   56          25
```

R Code

```
# set.seed(123)
```

```r
create_employee_dataset <- function(num_employees) {
  positions <- list(
    "Staff" = c(3000, 5000, 1, 5),
    "Supervisor" = c(5000, 8000, 5, 10),
    "Manager" = c(8000, 12000, 10, 15),
    "Director" = c(12000, 15000, 15, 25)
  )

  data <- data.frame(ID_Number = integer(),
                     Position = character(),
                     Salary = numeric(),
                     Age = integer(),
                     Experience = integer(),
                     stringsAsFactors = FALSE)

  for (i in 1:num_employees) {
    id_number <- sample(10000:99999, 1)
    position <- sample(names(positions), 1)
    salary <- sample(positions[[position]][1]:positions[[position]][2], 1)
    experience <- sample(positions[[position]][3]:positions[[position]][4], 1)
    age <- experience + sample(22:35, 1)

    data <- rbind(data, data.frame(ID_Number = id_number,
                                   Position = position,
                                   Salary = salary,
                                   Age = age,
                                   Experience = experience,
                                   stringsAsFactors = FALSE))
  }

  return(data)
}

df <- create_employee_dataset(20)
print(df)
```

```
   ID_Number   Position Salary Age Experience
1      66619 Supervisor   5556  28          5
2      59254      Staff   3196  31          2
3      59102      Staff   3832  36          3
4      11612   Director  12313  50         20
5      69619      Staff   4084  33          4
6      58685      Staff   3347  24          1
7      27222 Supervisor   6591  41          6
8      93174   Director  14285  49         23
9      88463   Director  14083  50         21
10     12747    Manager   9917  39         11
11     69690    Manager   9798  36         10
12     12570    Manager  11047  38         11
```

```
13     38455   Director  12748  56          23
14     19460    Manager  11398  36          11
15     31781      Staff   4460  39           5
16     70998    Manager   9173  36          12
17     36248    Manager   8391  47          14
18     48001 Supervisor   5292  41           8
19     27244 Supervisor   5804  41          10
20     75786   Director  12490  38          16
```

3.4.2 Filtering Data

We can use functions and loops to filter employees based on salary or experience levels.

Python Code

```python
def filter_high_salary(df, threshold=10000):
    return df[df['Salary'] > threshold]

high_salary_df = filter_high_salary(df, 10000)
print(high_salary_df)
```

```
    ID_Number  Position  Salary  Age  Experience
0       81265  Director   14129   43          15
1       70156  Director   13739   47          16
2       51249  Director   14322   49          22
6       10658  Director   13009   55          23
7       71600  Director   13287   51          19
8       38298  Director   13106   55          22
10      25628  Director   13910   49          21
12      20423  Director   14712   57          23
15      42241   Manager   10318   34          12
18      53829  Director   12036   50          25
19      84677  Director   13286   56          25
```

3.4.3 Aggregating Data

Using loops and functions, we can compute key statistics (exp: Average Salary and Experience) for different employee groups.

Python Code

```python
def compute_averages(df):
    return df.groupby("Position").agg({
        "ID_Number": "first",   # "Take one example ID"
        "Salary": "mean",
        "Age": "mean",
        "Experience": "mean"
    }).reset_index().round(2)
```

```
avg_stats = compute_averages(df)
print(avg_stats)
```

```
     Position  ID_Number    Salary    Age  Experience
0    Director      81265  13553.60  51.20       21.10
1     Manager      74338   9844.67  37.33       12.00
2       Staff      20685   3642.67  31.67        3.00
3  Supervisor      76138   6151.75  36.75        7.25
```

3.4.4 Determine Data

Loops can be used to determine the highest-paid employee in each position category.

Python Code

```
def top_earners(df):
    return df.loc[df.groupby("Position")["Salary"].idxmax()]

top_paid_employees = top_earners(df)
print(top_paid_employees)
```

```
    ID_Number    Position  Salary  Age  Experience
12      20423    Director   14712   57          23
15      42241     Manager   10318   34          12
11      43457       Staff    3955   30           5
13      76416  Supervisor    6719   35           9
```

Part II

Data Wrangling

Chapter 4

Data Gathering

4.1 What is Data Gathering?

Data gathering is the essential process of collecting data from various sources to empower your business with effective data-driven decision-making. Accurate and efficient data gathering provides the foundation for data analytics and business intelligence, enabling you to understand, visualize, and act on your data.

However, many businesses face the challenge of holding data in multiple locations with different formats. This can make it difficult to draw accurate comparisons, gain insights, and make important decisions based on your data gathering efforts.

The solution to this problem is data integration.

Chapter 5

Data Cleaning

Chapter 6

Data Transformation

Part III

EDA

Chapter 7

Descriptive Statistics

Chapter 8

Statistical Metrics

Chapter 9

Basic Visualizations

Chapter 10

Interactive Visualizations

Part IV

Applied DSP

Chapter 11

Data Science Automation

Chapter 12

Testing and Debugging