SINGAPORE UNIVERSITY OF SOCIAL SCIENCES

ANL252 Python for Data Analytics

STUDY GUIDE

School of Business

Course Development Team

ISBN 9789814873925

Educational Technology & Production
Singapore University of Social Sciences
463 Clementi Road
Singapore 599494

How to cite this Study Guide (APA):
Wu, K. Y. (2021). *ANL252 Python for data analytics (study guide)*. Singapore University of Social Sciences.

Release V1.0

Build S1.0.5, T1.5.21

Table of Contents

Course Guide

1. Welcome.......... CG-2
2. Course Description and Aims.......... CG-3
3. Learning Outcomes.......... CG-5
4. Learning Material.......... CG-6
5. Assessment Overview.......... CG-8
6. Course Schedule.......... CG-10
7. Learning Mode.......... CG-11

Study Unit 1: Introduction to Python Programming

Learning Outcomes.......... SU1-2

Overview.......... SU1-3

Chapter 1: Python Programming Environment.......... SU1-4

Chapter 2: Basic Arithmetic and Variables.......... SU1-14

Chapter 3: Print and Input.......... SU1-22

Chapter 4: If-elif-else-Conditions.......... SU1-34

Chapter 5: Loops.......... SU1-44

Summary.......... SU1-58

Formative Assessment.......... SU1-59

References.......... SU1-68

Study Unit 2: Data Types and Functions

Learning Outcomes........ SU2-2

Overview........ SU2-3

Chapter 1: Tuples, Lists, Dictionaries........ SU2-4

Chapter 2: Integrated Methods and Functions........ SU2-48

Chapter 3: User-defined Functions........ SU2-62

Chapter 4: Modules, Packages and Libraries........ SU2-67

Summary........ SU2-74

Formative Assessment........ SU2-75

References........ SU2-87

Study Unit 3: Arrays and Plots

Learning Outcomes........ SU3-2

Overview........ SU3-3

Chapter 1: Introduction to JupyterLab........ SU3-4

Chapter 2: Array Management with NumPy........ SU3-14

Chapter 3: Plotting with Matplotlib........ SU3-29

Summary........ SU3-38

Formative Assessment........ SU3-39

References........ SU3-49

Study Unit 4: Data Management

Learning Outcomes........ SU4-2

Overview.. SU4-3

Chapter 1: Import Data.. SU4-4

Chapter 2: Data Selection.. SU4-12

Chapter 3: Merge DataFrames.. SU4-26

Chapter 4: Missing Data and Outliers.. SU4-38

Chapter 5: Data Modification.. SU4-55

Summary.. SU4-65

Formative Assessment.. SU4-66

References.. SU4-77

Study Unit 5: Data Analytics in Python

Learning Outcomes.. SU5-2

Overview.. SU5-3

Chapter 1: Introduction to Scikit-Learn.. SU5-4

Chapter 2: Clustering.. SU5-35

Chapter 3: Decision Trees.. SU5-53

Summary.. SU5-74

Formative Assessment.. SU5-75

References.. SU5-85

Study Unit 6: Basic SQL in Python

Learning Outcomes.. SU6-2

Overview.. SU6-3

Chapter 1: Introduction to SQL and SQLite3.. SU6-4

Chapter 2: Data Query........ SU6-18
Chapter 3: Joining Tables........ SU6-34
Chapter 4: Grouping Data........ SU6-48
Chapter 5: Editing Data........ SU6-56
Summary........ SU6-73
Formative Assessment........ SU6-74
References........ SU6-84

List of Tables

Table 1.1 Python Arithmetic Operators.. SU1-15

Table 1.2 Types of Variable in Python... SU1-20

Table 1.3 List of Some Escape Sequences Available in Python............................ SU1-28

Table 1.4 Examples of Relational Operation... SU1-34

Table 1.5 List of Relational Operators... SU1-35

Table 1.6 List of Logical Operators.. SU1-36

Table 1.7 List of Some Common Exceptions... SU1-53

Table 2.1 Built-in Functions of Python.. SU2-48

Table 2.2 Built-in Methods of Python.. SU2-51

Table 2.3 Some Common Packages for Data Analytics in Python...................... SU2-73

Table 3.1 Most Common NumPy Functions... SU3-22

Table 4.1 Most Common Data File Formats and the Corresponding Reader in pandas.. SU4-5

Table 5.1 Most Common Algorithms Available in scikit-learn.............................. SU5-4

Table 5.2 Parameters of the KMeans Estimator.. SU5-38

Table 5.3 Parameters of the `DecisionTreeClassifier` and `DecisionTreeRegressor` Estimators.. SU5-57

Table 5.4 Parameters of the plot_tree Function.. SU5-69

Table 6.1 List of Some Available Modes of the `open()` Function........................ SU6-6

Table 6.2 Operators in the `WHERE` Clause.. SU6-24

Table 6.3 List of Aggregate Functions in SQL.. SU6-48

List of Figures

Figure 1.1 Python Website: Download the Latest Version.................................... SU1-4

Figure 1.2 Python Installation on Windows: Check the Box to Add Python to PATH.. SU1-5

Figure 1.3 Download Website of Atom for Windows.. SU1-6

Figure 1.4 Finding Python in the Start Menu... SU1-7

Figure 1.5 Python Interpreter.. SU1-7

Figure 1.6 Searching for Windows PowerShell and Command Prompt............. SU1-8

Figure 1.7 Launching Python in Windows PowerShell... SU1-9

Figure 1.8 Execute a Simple Python Code... SU1-10

Figure 1.9 Quit the Python Interpreter.. SU1-10

Figure 1.10 Writing Python Code with Atom... SU1-11

Figure 1.11 A Comment Line in Python.. SU1-12

Figure 1.12 Executing a Python Scripts written in Atom Using PowerShell... SU1-12

Figure 1.13 Simple Calculations with Python... SU1-14

Figure 1.14 Assigning Values to Variables for Operations.................................. SU1-19

Figure 1.15 Checking Variable Type.. SU1-21

Figure 1.16 Simple Use of the print() Function... SU1-22

Figure 1.17 Printing Variables and Their Calculations.. SU1-23

Figure 1.18 Printing Formatted String.. SU1-24

Figure 1.19 Output of Incomplete String Formatting Syntax............................. SU1-24

Figure 1.20 Simple Usage of the `.format()` Method.. SU1-25

Figure 1.21 Multiple Usage of `.format()` Method.. SU1-25

Figure 1.22 Syntax Error Caused by Quotation Marks within a string............. SU1-26

Figure 1.23 Printing Quotation Marks within a String....................................... SU1-27

Figure 1.24 Printing Quotation Marks using Escape Sequence........................... SU1-27

Figure 1.25 Erroneous Line Breaks within a String... SU1-27

Figure 1.26 Line Breaks Created by Escape Sequence `\n`.................................... SU1-28

Figure 1.27 Example of Using `input()` Function.. SU1-31

Figure 1.28 Convert Input Value to Integer and Float... SU1-32

Figure 1.29 `if`-Statement Example with `True` Condition.................................. SU1-39

Figure 1.30 `if`-Statement Example with `False` Condition................................ SU1-39

Figure 1.31 `if-else`-Statement Example.. SU1-40

Figure 1.32 Example of `if-elif-else`-Statement.. SU1-42

Figure 1.33 Repeating the Same Code for Two Individuals to Enter................. SU1-45

Figure 1.34 Entering Names and Scores Repeatedly by `while`-Loops............. SU1-46

Figure 1.35 Entering Names and Scores Repeatedly by `for`-Loops.................. SU1-49

Figure 1.36 Breaking from Loops Using `break`.. SU1-51

Figure 1.37 Using Exception for Error Handling.. SU1-54

Figure 1.38 Using `while`-loop for User Input If Error Occurs........................... SU1-55

Figure 1.39 The Complete Example Program of Study Unit 1............................ SU1-57

Figure 2.1 Definition of Tuples.. SU2-5

Figure 2.2 Subsetting Tuples.. SU2-6

Figure 2.3 Subsetting of Tuples with Negative Indices... SU2-7

Figure 2.4 Subsetting Tuples with "Open End" Indexing...................................... SU2-8

Figure 2.5 Erroneous Modification of Tuples.. SU2-9

Figure 2.6 Erroneous Adding of Elements to Tuples... SU2-9

Figure 2.7 Concatenation of Two Tuples.. SU2-10

Figure 2.8 Adding Elements to Tuples by Concatenation................................... SU2-10

Figure 2.9 Length of Tuples and Index Out of Range.. SU2-12

Figure 2.10 Subsetting the Last Element of Tuples... SU2-12

Figure 2.11 Printing Tuple Elements by `for`-Loops... SU2-14

Figure 2.12 Creating Lists in Python.. SU2-15

Figure 2.13 Subsetting Lists... SU2-16

Figure 2.14 Editing Single Item in a List.. SU2-18

Figure 2.15 Editing Multiple Items in a List.. SU2-19

Figure 2.16 Replacing More Values than the Specified Indices.......................... SU2-19

Figure 2.17 Concatenating Lists.. SU2-21

Figure 2.18 Concatenating Lists with Different Data Types................................ SU2-22

Figure 2.19 Merging Two Lists by Keeping Them as Lists in the New
One.. SU2-23

Figure 2.20 Extracting Elements from a Merged List... SU2-24

Figure 2.21 Printing List Elements Line by Line to the Screen........................... SU2-26

Figure 2.22 Printing Elements of Multiple Lists Using Formatted String.......... SU2-27

Figure 2.23 Rearranging Data Storage in Lists.. SU2-28

Figure 2.24 Printing Data of a List Using Formatted String............................... SU2-30

Figure 2.25 User's Input to a List.. SU2-31

Figure 2.26 Defining a Dictionary... SU2-36

Figure 2.27 Extracting Value from a Dictionary.. SU2-37

Figure 2.28 Extracting Keys and Values from a Dictionary................................ SU2-38

Figure 2.29 Store Keys and Values from a Dictionary in Lists............................ SU2-38

Figure 2.30 Print All the Keys from a Dictionary Line by Line.......................... SU2-39

Figure 2.31 Print All the Keys and Values from a Dictionary Line by Line....... SU2-39

Figure 2.32 Extracting Items from a Dictionary.. SU2-40

Figure 2.33 Print Items from a Dictionary after Converting it to a List............. SU2-41

Figure 2.34 Change One Value in a Dictionary... SU2-42

Figure 2.35 Add Items to Every Dictionary Key and Convert Values to Lists... SU2-43

Figure 2.36 Print Items from a Dictionary after Converting to Lists Using `for`-loops... SU2-43

Figure 2.37 Changing a key of a Dictionary.. SU2-44

Figure 2.38 Adding a New Key to a Dictionary.. SU2-45

Figure 2.39 Merging Two Dictionaries Using the v3.5+ Option.......................... SU2-46

Figure 2.40 Merging Two Dictionaries Using the v3.9+ Option.......................... SU2-46

Figure 2.41 Computing Statistics of a Dictionary's Values.................................. SU2-50

Figure 2.42 Replacing Double Spacing by Single Spacing.................................. SU2-57

Figure 2.43 Capitalise Each Word in Strings.. SU2-58

Figure 2.44 Switching Surname and First name in Strings................................. SU2-59

Figure 2.45 Sorting Dictionary by Its Keys.. SU2-60

Figure 2.46 Defining Functions.. SU2-64

Figure 2.47 Sorting Dictionary Using User-Defined Functions........................... SU2-65

Figure 2.48 Integrate Entire Package in a Program.. SU2-69

Figure 2.49 Integrate One Module from a Package.. SU2-70

Figure 2.50 Installing Package "numpy" with pip.. SU2-71

Figure 2.51 Installing Package "matplotlib" with pip3... SU2-72

Figure 3.1 The Official Website for Jupyter.. SU3-5

Figure 3.2 Start Installing JupyterLab... SU3-6

Figure 3.3 Installation of Jupyter Completed.. SU3-6

Figure 3.4 Starting the JupyterLab.. SU3-7

Figure 3.5 Start-Up Page of JupyterLab... SU3-8

Figure 3.6 Blank Python Script.. SU3-8

Figure 3.7 Running a Python Script in JupyterLab.. SU3-9

Figure 3.8 Inserting a New Cell in JupyterLab... SU3-9

Figure 3.9 Functions Included in the Edit Menu of JupyterLab.......................... SU3-10

Figure 3.10 Saving Python Program in JupyterLab... SU3-11

Figure 3.11 Changing the Cell Type in JupyterLab... SU3-11

Figure 3.12 Editing a Markdown Cell in JupyterLab... SU3-12

Figure 3.13 A Finalised Markdown Cell with a Comment in JupyterLab.......... SU3-12

Figure 3.14 A Finalised Markdown Cell with Ordinary Text in JupyterLab... SU3-13

Figure 3.15 Installing Package "numpy"... SU3-15

Figure 3.16 Creating an Array with NumPy.. SU3-16

Figure 3.17 Subsetting a Column from a NumPy Array..................................... SU3-18

Figure 3.18 Subsetting a Sub-Array from a NumPy Array................................. SU3-19

Figure 3.19 Creating a Boolean Mask of a NumPy Array................................... SU3-19

Figure 3.20 Subsetting a NumPy Array Based on a Boolean Mask.................... SU3-20

Figure 3.21 Extracting Information on the Characteristics of a NumPy Array.. SU3-21

Figure 3.22 Calculate Column Statistics of a NumPy Array............................... SU3-27

Figure 3.23 Importing "matplotlib.pyplot" into the Program............................ SU3-31

Figure 3.24 Program for Creating a Line Plot.. SU3-32

Figure 3.25 Line Plot Generated by "matplotlib.pyplot".................................... SU3-32

Figure 3.26 Creating Data Array for Subsequent Plotting.................................. SU3-34

Figure 3.27 Program for Creating a Histogram... SU3-34

Figure 3.28 Historam Generated by “matplotlib.pyplot”.................................... SU3-35

Figure 3.29 Program for Creating a Scatter Plot.. SU3-36

Figure 3.30 Scatter Plot Generated by “matplotlib.pyplot”................................ SU3-37

Figure 4.1 Importing Data with pandas.. SU4-9

Figure 4.2 Printing Entire Imported Dataset.. SU4-9

Figure 4.3 Printing the Head and Tail of the Imported Dataset by `display()`.. SU4-10

Figure 4.4 Select Columns from a DataFrame by Variable Names.................... SU4-13

Figure 4.5 Select a Single Column from a DataFrame and Save it as NumPy Array.. SU4-13

Figure 4.6 Select Rows from a DataFrame by Random Row Indices................. SU4-14

Figure 4.7 Setting the Values of a Column as Row Index.................................. SU4-16

Figure 4.8 Resetting Row Index of a DataFrame.. SU4-16

Figure 4.9 Setting a NumPy Array as Row Index.. SU4-17

Figure 4.10 Selecting Rows by Single Row Label... SU4-18

Figure 4.11 Selecting Rows by Multiple Row Labels... SU4-18

Figure 4.12 Selecting Cells Using Indices... SU4-20

Figure 4.13 Selecting Cells Using Row and Column Labels............................... SU4-21

Figure 4.14 Selecting Cells by Row Indices and Column Labels........................ SU4-21

Figure 4.15 Selecting Cells by Row Labels and Column Indices........................ SU4-22

Figure 4.16 Selecting Cells by a Boolean Mask.. SU4-23

Figure 4.17 Selecting Cells by Chaining Two Boolean Masks.............................. SU4-24

Figure 4.18 Selecting Cells by Chaining Multiple Boolean Masks...................... SU4-25

Figure 4.19 Concatenating Two Datasets with Different Rows but Identical Variables.. SU4-26

Figure 4.20 Appending Two DataFrames.. SU4-27

Figure 4.21 Concatenating Two Datasets with Different Columns but Identical Observations.. SU4-29

Figure 4.22 Outer Join Two Datasets with Some Common Variables................. SU4-29

Figure 4.23 Inner Join Two Datasets with Some Common Variables.................. SU4-30

Figure 4.24 Outer Join Two Datasets with Some Common Observations.......... SU4-31

Figure 4.25 Inner Join Two Datasets with Some Common Observations........... SU4-31

Figure 4.26 Outer Join Two Datasets with Different Shapes................................ SU4-32

Figure 4.27 Inner Join Two Datasets with Different Shapes................................. SU4-32

Figure 4.28 Concatenating Two DataFrames with Different Shapes by Outer Join.. SU4-34

Figure 4.29 Concatenating Two DataFrames with Different Shapes by Inner Join.. SU4-35

Figure 4.30 Concatenating Two DataFrames with Same Variables by Rows... SU4-35

Figure 4.31 Concatenating Two DataFrames with Same Observations by Columns... SU4-36

Figure 4.32 Declaring Specific Strings as Missing Values While Importing Data... SU4-40

Figure 4.33 Counting the Number of Missing Values in Each Variable............. SU4-43

Figure 4.34 Identifying Observations with Missing Values................................ SU4-44

Figure 4.35 Selecting Observations with Missing Values from a DataFrame.. SU4-44

Figure 4.36 Dropping Observations with Missing Values Using `.drop()` Method.. SU4-47

Figure 4.37 Dropping Observations with Missing Values Using `.dropna()` Method.. SU4-48

Figure 4.38 Replacing Missing Values by 0 in the Entire DataFrame................. SU4-49

Figure 4.39 Replacing Missing Values by "Unknown" in Specific Columns.. SU4-50

Figure 4.40 Computing Criteria for Outlier Detection in a Numeric Variable.. SU4-52

Figure 4.41 Selected Outlier Observations from a DataFrame............................. SU4-53

Figure 4.42 Selected Non-Outlier Observations from a DataFrame.................... SU4-54

Figure 4.43 Sorting a DataFrames by Two Variables.. SU4-56

Figure 4.44 Discretising a Numeric Variable into Bins.. SU4-58

Figure 4.45 Computing the Mean of All Numeric Columns for Grouped Data.. SU4-60

Figure 4.46 Computing the Mean of Selected Columns for Grouped Data....... SU4-60

Figure 4.47 Log-Transformation of a Numeric Variable...................................... SU4-62

Figure 4.48 Standardisation of a Numeric Variable... SU4-63

Figure 4.49 Normalisation of a Numeric Variable.. SU4-64

Figure 5.1 Importing Modules and Functions from scikit-learn.......................... SU5-7

Figure 5.2 Removing Missing Data from DataFrame.. SU5-11

Figure 5.3 Listing Out Unique Categories of a Variable...................................... SU5-12

Figure 5.4 Creating a Dictionary with Old and New Category Labels.............. SU5-13

Figure 5.5 Reducing the Number of Categories for a Categorical Variable....... SU5-13

Figure 5.6 Reducing the Number of Categories for a Categorical Variable....... SU5-14

Figure 5.7 Cross Tabulation of Two Categorical Variables.................................. SU5-16

Figure 5.8 Numeric Labels of a Categorical Variable... SU5-17

Figure 5.9 Discretising an Ordered Numeric Categorical Variable..................... SU5-18

Figure 5.10 Selecting Relevant Variables.. SU5-20

Figure 5.11 Renaming Selected Variables... SU5-20

Figure 5.12 Creating Dummy Variables from Categorical Variables................. SU5-22

Figure 5.13 Selecting Numeric Variables for Normalisation............................... SU5-25

Figure 5.14 Normalising Numeric Variables.. SU5-26

Figure 5.15 Concatenating Normalised Variables with the Original DataFrame.. SU5-26

Figure 5.16 Training Dataset after Split from the Original DataFrame............... SU5-29

Figure 5.17 Testing Dataset after Split from the Original DataFrame................. SU5-29

Figure 5.18 Saving the Row Indices in Python Lists.. SU5-31

Figure 5.19 Slicing Training and Testing Datasets for the Independent and the Target Variables.. SU5-32

Figure 5.20 Training Dataset for the Target Variable.......... SU5-32

Figure 5.21 Testing Dataset for the Target Variable.......... SU5-33

Figure 5.22 Training Dataset for the Independent Variables.......... SU5-33

Figure 5.23 Testing Dataset for the Independent Variables.......... SU5-33

Figure 5.24 Dataset for the Dependent Variable.......... SU5-34

Figure 5.25 Dataset for the Independent Variables.......... SU5-34

Figure 5.26 Create a DataFrame with Only Normalised Input Variables.......... SU5-41

Figure 5.27 Calculating Inertia for the Elbow Method.......... SU5-42

Figure 5.28 Elbow Method to Determine the Optimal Number of Clusters.......... SU5-42

Figure 5.29 Fitting K-Means Clustering.......... SU5-43

Figure 5.30 Predicting Classification of the Data.......... SU5-43

Figure 5.31 Converting the Cluster Index Array to pandas DataFrame.......... SU5-46

Figure 5.32 Merging the Original DataFrame with the Cluster Index.......... SU5-46

Figure 5.33 Cross-Tabulation of `income` and the Predicated Classification.......... SU5-46

Figure 5.34 Cross-Tabulation of `workclass` and the Predicated Classification.......... SU5-47

Figure 5.35 Cross-Tabulation of `education` and the Predicated Classification.......... SU5-47

Figure 5.36 Cross-Tabulation of `race` and the Predicated Classification.......... SU5-48

Figure 5.37 Cross-Tabulation of Numeric Variables and the Predicated Classification........ SU5-48

Figure 5.38 Calculating the Silhouette Coefficient........ SU5-49

Figure 5.39 Reducing the Dimensionality of the Input Variable DataFrame........ SU5-50

Figure 5.40 Selecting PCA Data Based on the Predicted Classification........ SU5-51

Figure 5.41 Plotting of the K-Means Clustering Result........ SU5-51

Figure 5.42 Create a DataFrame without the Normalised Input Variables........ SU5-64

Figure 5.43 Fitting Decision Trees........ SU5-64

Figure 5.44 Predicting Classification of Training and Testing Datasets........ SU5-65

Figure 5.45 Prediction Performance on Training Data........ SU5-67

Figure 5.46 Prediction Performance on Testing Data........ SU5-67

Figure 5.47 Confusion Matrix........ SU5-68

Figure 5.48 Preparing Labels for Tree Plot........ SU5-71

Figure 5.49 Plotting Decision Tree with `tree.plot_tree`........ SU5-71

Figure 5.50 Decision Tree Plot........ SU5-72

Figure 6.1 Importing sqlite3 and pandas........ SU6-5

Figure 6.2 Program to Input Data into a .csv File........ SU6-9

Figure 6.3 Interface to Input Data in JupyterLab........ SU6-10

Figure 6.4 Print the Records in a .csv File to the Screen Line by Line........ SU6-11

Figure 6.5 Establish a Connection between Python and a Database........ SU6-13

Figure 6.6 Import a.csv file Dataset as a pandas DataFrame.............................. SU6-14

Figure 6.7 Export pandas DataFrame to Database by SQL................................ SU6-14

Figure 6.8 Select a Table from the Database.. SU6-16

Figure 6.9 Fetch One Record for Printing.. SU6-16

Figure 6.10 Fetch All Records for Printing... SU6-16

Figure 6.11 Re-Fetch Records after Applying `fetchone()` or `fetchall()`.. SU6-17

Figure 6.12 Extract Column Names of a Table by the `.description` Attribute... SU6-19

Figure 6.13 Generate Variable List of a Table.. SU6-19

Figure 6.14 Execute an SQL Statement Stored in a String Variable..................... SU6-20

Figure 6.15 Convert the Result of an SQL Query to a pandas DataFrame......... SU6-21

Figure 6.16 Sort Students by Their Birthdays and Nationalities......................... SU6-23

Figure 6.17 Select only Analytics Students from the Table `students`............... SU6-26

Figure 6.18 Select only Analytics Students with ID number between 5 And 10.. SU6-26

Figure 6.19 Select only Analytics Students with ID number not between 5 And 10.. SU6-27

Figure 6.20 Select Analytics Students not from Singapore or China.................. SU6-27

Figure 6.21 Select Analytics Students with First Names that Start with "M"... SU6-28

Figure 6.22 Select Analytics Students with First Names that Contain "ar"....... SU6-28

Figure 6.23 Select Analytics Students with First Names that End with "s"....... SU6-28

Figure 6.24 Select Analytics Students with First Names that Start with "M" and End with "l".. SU6-29

Figure 6.25 Select Analytics Students with Last Names that Have the Pattern "Ta_".. SU6-29

Figure 6.26 Select Students with No Birthday Record.. SU6-29

Figure 6.27 Select Students with Non-Missing Birthday Records....................... SU6-30

Figure 6.28 Select Data of Non-Analytics Students from Certain Columns of a Table.. SU6-31

Figure 6.29 Concatenate Items of a Tuple to a String.. SU6-31

Figure 6.30 Select Certain Columns of a Table by a Variable List String............ SU6-32

Figure 6.31 Create a New Table Called `grades` in the Database........................ SU6-36

Figure 6.32 Contents of the Table `grades`.. SU6-36

Figure 6.33 Inner Join the Tables `students` and `grades` by the `ON` Keyword.. SU6-37

Figure 6.34 Inner Join the Tables `students` and `grades` by the `USING` Keyword.. SU6-38

Figure 6.35 Left Join the Tables `students` and `grades`.................................... SU6-40

Figure 6.36 Sort the Left Joined Table by the Course Code and the Students' Last Name.. SU6-41

Figure 6.37 Select Records with Missing Data in `grade`.................................... SU6-41

Figure 6.38 Cross Join the Tables `students` and `grades`................................. SU6-43

Figure 6.39 Generate Variable List from Both Tables with Aliases...................... SU6-45

Figure 6.40 Outer Join the Tables `students` and `grades` (1)............................ SU6-46

Figure 6.41 Outer Join the Tables `students` and `grades` (2)............................ SU6-46

Figure 6.42 Count the Number of Students in Each Programme........................ SU6-49

Figure 6.43 Sort the Nationalities of the Students by Their Counts.................... SU6-50

Figure 6.44 Group the Students by their Study Programmes and Nationalities.. SU6-50

Figure 6.45 Calculate Average Grade of Students from Different Birthyears.. SU6-52

Figure 6.46 Calculate Average Grade of Different Programmes in Different Courses.. SU6-53

Figure 6.47 Selected Courses with `NumStudents >=` 5 Grouped by Study Programme.. SU6-53

Figure 6.48 Listing Students with `AverageGrade >= 40`................................ SU6-54

Figure 6.49 Insert Multiple New Records for a Student into the Table `grades`.. SU6-57

Figure 6.50 Insert a New Record into the Table `students` without Variable List.. SU6-57

Figure 6.51 Update the Value of a Selected Record in the Variable `Grade`........ SU6-59

Figure 6.52 Delete Records with ID Number >=22 from the Table `grades`.. SU6-61

Figure 6.53 Add a New Column called `email` to the Table `students`............ SU6-63

Figure 6.54 The Table `students` after a New Column `email` Being Added.. SU6-63

Figure 6.55 Update Values in the Column `email`.. SU6-64

Figure 6.56 Rename the Column from `email` to `Email`..................................... SU6-65

Figure 6.57 Add a New Column Called `Name` to the Table `students`............. SU6-66

Figure 6.58 Concatenate Last Name and First Name as a New Variable........... SU6-67

Figure 6.59 Create a New Table named `students2`... SU6-67

Figure 6.60 Query Data from `students` and Insert Them Into `students2`.. SU6-68

Figure 6.61 Output of students2 after Inserting the Data.................................... SU6-68

Figure 6.62 Check on Existing Tables in the Database... SU6-71

Figure 6.63 Dropping Table `students` from the Database................................ SU6-71

Figure 6.64 Renaming `students2` to `student`.. SU6-72

Figure 6.65 Commit Changes of the Database to Physical File and Close Connection.. SU6-72

List of Lesson Recordings

Introduction to Python Programming........ SU1-4

Basic Arithmetic and Variables in Python........ SU1-14

Print and Input in Python........ SU1-22

If-elif-else-Conditions in Python........ SU1-34

Loops in Python........ SU1-44

Python Tuples........ SU2-4

Python Lists........ SU2-14

Python Dictionaries........ SU2-35

Integrated Methods and Functions in Python........ SU2-48

User-defined Functions in Python........ SU2-62

Modules, Packages and Libraries in Python........ SU2-67

Introduction to JupyterLab........ SU3-4

Array Management with NumPy........ SU3-14

Plotting with matplotlib........ SU3-29

Import Data in pandas........ SU4-4

Data Selection in pandas........ SU4-12

Merge DataFrames in pandas........ SU4-26

Missing Data and Outliers in pandas DataFrames........ SU4-38

Data Modification in pandas.. SU4-55

Introduction to Scikit-Learn.. SU5-4

Data Preparation for Analytics Algorithms of scikit-learn (1/2)........................... SU5-9

Data Preparation for Analytics Algorithms of scikit-learn (2/2)........................... SU5-9

Introduction and Fitting of K-Means Clustering by scikit-learn......................... SU5-35

Explore & Evaluate K-Means Clustering Models by scikit-learn........................ SU5-44

Introduction and Fitting of Decision Trees by scikit-learn.................................. SU5-53

Evaluate Decision Trees by scikit-learn.. SU5-66

Introduction to SQL.. SU6-4

Data Query with SQL.. SU6-18

Join Tables with SQL... SU6-34

Group Data with SQL.. SU6-48

Edit Data with SQL... SU6-56

Course Guide

Python for Data Analytics

1. Welcome

Presenter: Dr Karl Wu Ka Yui

This streaming video requires Internet connection. Access it via Wi-Fi to avoid incurring data charges on your personal mobile plan.

Click here to watch the video. [i]

Click here for the transcript.

Welcome to the course ***ANL252 Python for Data Analytics***, a 5 credit unit (CU) course.

This Study Guide will be your personal learning resource to take you through the course learning journey. The guide is divided into two main sections – the Course Guide and Study Units.

The Course Guide describes the structure for the entire course and provides you with an overview of the Study Units. It serves as a roadmap of the different learning components within the course. This Course Guide contains important information regarding the course learning outcomes, learning materials and resources, assessment breakdown and additional course information.

[i] https://d2jifwt31jjehd.cloudfront.net/ANL252/IntroVideo/ANL252_Intro_Video.mp4

2. Course Description and Aims

The course provides foundational knowledge and skills of Python programming, which enables students to develop programs for data preparation, data management, and data visualisation to carry out data analytics tasks such as clustering, decision tree, etc. Students also acquire skills to explore and find patterns in datasets using ready-to-use Python codes that can be modified to suit individual needs. Furthermore, this course introduces the application of SQL for querying data from database within any Python code for data analytics purposes. Since this course is designed to help students with little prior exposure to programming, it will focus on breadth rather than depth.

Course Structure

This course is a 5-credit unit course presented over six weeks.

There are six Study Units in this course. The following provides an overview of each Study Unit.

Study Unit 1 – Introduction to Python Programming

This unit takes the first steps in Python programming, including variables, data types, operators, formatted printing, and user input. It also introduces the conditional statement and loops, two types of control flow that change the behaviour of a program dynamically.

Study Unit 2 – Data Types and Functions

This unit establishes three compound built-in data types in Python: tuples, lists, and dictionaries. Compound data structures organise and store data in a way that they can be accessed and worked with efficiently. The type of the compound data also defines the relationship between the data and the operations that can be performed on them. Furthermore, this unit covers the application of functions, methods, packages, and modules and how they can be integrated in the program.

Study Unit 3 – Arrays and Plots

This study unit introduces two Python packages: NumPy and matplotlib. NumPy is the fundamental package for efficient scientific computing with Python. The students first learn to create and subset NumPy arrays, before getting on to generate statistics on the data stored in an array by the integrated NumPy functions. Furthermore, this unit illustrates the use of the matplotlib package for data visualisation, including the functionality for plotting and customising basic charts of data analytics.

Study Unit 4 – Data Management

This unit establishes the pandas DataFrame as the key data structure for analytics in Python. It demonstrates the process of creating Python programs for importing data from external sources, indexing, and querying data from a DataFrame. It also focuses on merging multiple DataFrames efficiently, identifying and dealing with missing data and outliers, sorting, grouping, and transforming data, as well as discretising numeric variables to bins.

Study Unit 5 – Data Analytics in Python

This unit covers the application of k-means clustering and decision trees in Python programming using the scikit-learn library. The specific preparation of DataFrames due to the different requirements of the analytics algorithms is discussed in the first step, followed by the implementation of the two techniques in Python programs, including parameter settings, presentation, and visualisation of the results.

Study Unit 6 – Basic SQL in Python

This unit describes the usage of SQL to query data from databases in Python programs. The students learn how to create flexible SQL statements for data query such as selecting data, sorting data, grouping data, merging tables, and modifying data using Python programming skills. Another focus of this unit is to use Python coding to present and convert output of data query as pandas DataFrames for further analytical process.

3. Learning Outcomes

Knowledge & Understanding (Theory Component)

By the end of this course, you will be able to:

- Differentiate the various aspects of Python programming.
- Discuss how Python manages packages, modules, functions, etc.
- Explain the operations on arrays and datasets.

Key Skills (Practical Component)

By the end of this course, you will be able to:

- Design Python programmes for performing data analytics.
- Employ logic control flows in Python programmes.
- Prepare data for analysis using Python programming.
- Analyse data using appropriate tools and techniques with Python programming.

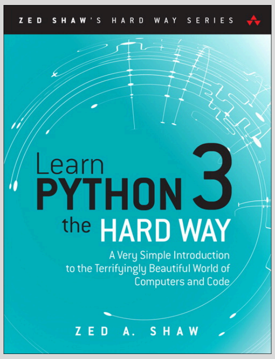
4. Learning Material

To complete the course, you will need the following learning material(s):

Required Textbook(s)

Shaw, Z. A. (2017). *Learn python 3 the hard way*. Addison-Wesley Professional.

If you are enrolled into this course, you will be able to access the eTextbooks here:


ZED SHAW'S HARD WAY SERIES
Learn PYTHON 3 the HARD WAY
A Very Simple Introduction to the Terrifyingly Beautiful World of Computers and Code
ZED A. SHAW

To launch eTextbook, you need a VitalSource account which can be created via Canvas (iBookStore), using your SUSS email address. Access to adopted eTextbook is restricted by enrolment to this course.

Recommended Study Material(s) for Learning Activities (Optional)

Website(s):

The Python documentation. https://docs.python.org/

JupyterLab Documentation. https://jupyterlab.readthedocs.io/en/stable/

NumPy user guide. https://numpy.org/

matplotlib pyplot overview. https://matplotlib.org/stable/api/pyplot_summary.html

pandas documentation. https://pandas.pydata.org/pandas-docs/stable/index.html scikit learn API reference. https://scikit-learn.org/stable/modules/classes.html

Learn Python. tutorialspoint. https://www.tutorialspoint.com/python/

Python Tutorial, SQL Tutorial. https://www.w3schools.com/

SQLite Tutorial. https://www.sqlitetutorial.net/

5. Assessment Overview

The overall assessment weighting for this course is as follows:

Assessment	Description	Weight Allocation
Assignment 1	Pre-Course Quiz 01	2%
	Pre-Class Quiz 01	2%
	Pre-Class Quiz 02	2%
Assignment 2	Tutor-Marked Assignment	18%
Assignment 3	Group-Based Assignment	20%
Participation	Participation during Seminar	6%
Examination	End-of-Course Assignment	50%
TOTAL		100%

The following section provides important information regarding Assessments.

Continuous Assessment:

There will be continuous assessment in the form of one pre-course quiz, two pre-class quizzes, one tutor-marked assignment (TMA) and one group-based assignment (GBA). In total, this continuous assessment will constitute 50 percent of overall student assessment for this course. The assignments are compulsory and are non-substitutable. These assignments will test conceptual understanding of both the fundamental and more advanced concepts and applications that underlie Python programming and its application in data analytics. It is imperative that you read through your Assignment questions and submission instructions before embarking on your Assignment.

Examination:

The end-of-course assignment will constitute the other 50 percent of overall student assessment and will test the ability in applying Python programming to solve data analytics related tasks such as data management and data analysis. All topics covered in the course outline will be examinable. To prepare for the exam, you are advised to review Specimen or Past Year Exam Papers available on Learning Management System.

Passing Mark:

To successfully pass the course, you must obtain a minimum passing mark of 40 percent for the combined continuous assessments. That is, students must obtain at least a mark of 40 percent for the combined assessments and also at least a mark of 40 percent for the end-of-course assessment. For detailed information on the Course grading policy, please refer to The Student Handbook (‘Award of Grades’ section under Assessment and Examination Regulations). The Student Handbook is available from the Student Portal.

Non-graded Learning Activities:

Activities for the purpose of self-learning are present in each study unit. These learning activities are meant to enable you to assess your understanding and achievement of the learning outcomes. The type of activities can be in the form of Formative Assessment, Quiz, Review Questions, Application-Based Questions or similar. You are expected to complete the suggested activities either independently and/or in groups.

6. Course Schedule

To pace yourself and monitor your study progress, pay special attention to your Course Schedule. It contains study-unit-related activities including Assignments, Self-Assessments, and Examinations. Please refer to the Course Timetable on the Student Portal for the most current Course Schedule.

Note: Always make it a point to check the Student Portal for announcements and updates.

7. Learning Mode

The learning approach for this course is structured along the following lines:

a. Self-study guided by the study guide units. Independent study will require ***at least 3 hours per week.***

b. Working on assignments, either individually or in groups.

c. Classroom Seminars (3 hours each session, 6 sessions in total).

iStudyGuide

You may be viewing the interactive StudyGuide (iStudyGuide), which is the mobile-friendly version of the Study Guide. The iStudyGuide is developed to enhance your learning experience with interactive learning activities and engaging multimedia. You will be able to personalise your learning with digital bookmarking, note-taking, and highlighting of texts if your reader supports these features.

Interaction with Instructor and Fellow Students

Flexible learning—learning at your own pace, space, and time—is a hallmark at SUSS, and we strongly encourage you to engage your instructor and fellow students in online discussion forums. Sharing of ideas through meaningful debates will help broaden your perspective and crystallise your thinking.

Academic Integrity

As a student of SUSS, you are expected to adhere to the academic standards stipulated in the Student Handbook, which contains important information regarding academic policies, academic integrity, and course administration. It is your responsibility to read and understand the information outlined in the Student Handbook prior to embarking on the course.

Study Unit 1

Introduction to Python Programming

Learning Outcomes

By the end of this unit, you should be able to:

1. Differentiate the various aspects of Python programming
2. Employ logic control flows in Python programmes

Overview

This study unit introduces the Python programming environment and the writing of Python programs with some foundation elements. We will also learn how to create different types of variables and how to assign values to them for further operations. Since input and output belong to the core of any computer program, we will learn how to create user input and construct formatted strings for printing as well. Also, we will cover the construction of Boolean expressions as conditional statements to control the behaviour of the program. Eventually, we will find out how to create finite loops to repeat routine instructions iteratively.

Chapter 1: Python Programming Environment

Lesson Recording

Introduction to Python Programming

1.1 Installation of Python and Atom

Visit the URL https://www.python.org/downloads/ and click on "Download Python [Version]", where [Version] is the latest version number of Python (Version 3.9.0 is the latest version when this Study Guide was being developed).

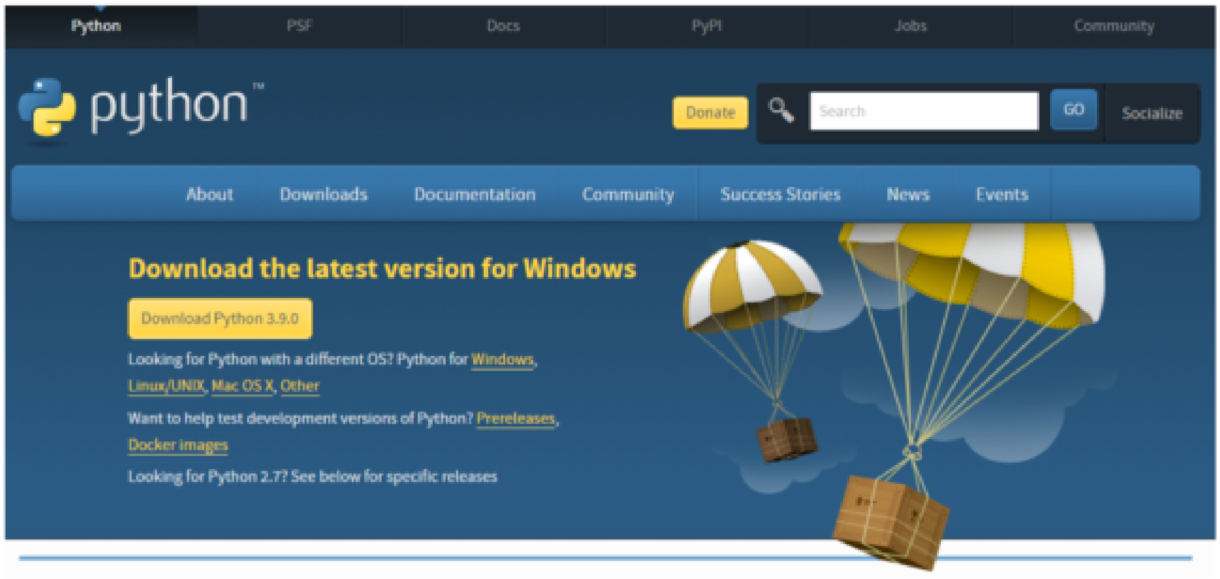
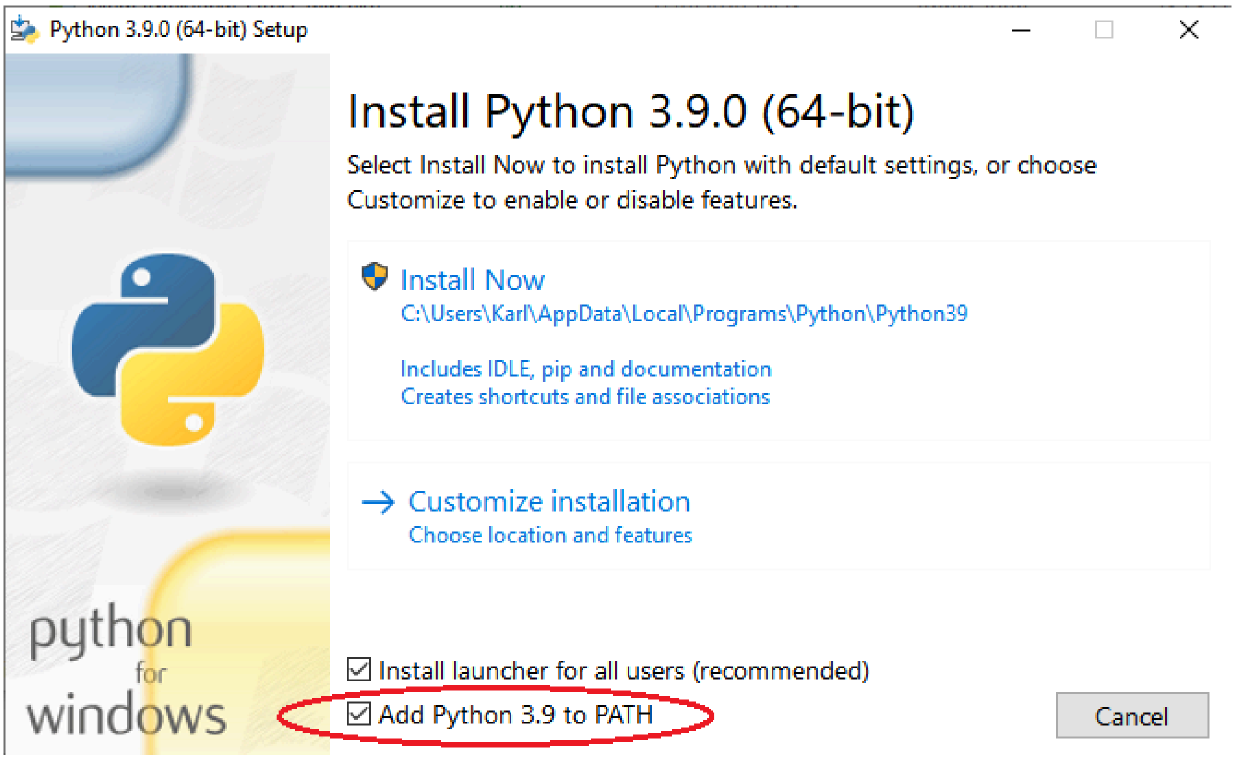

Python
PSF
Docs
PyPI
Jobs
Community
python
Donate
Search
GO
Socialize
About
Downloads
Documentation
Community
Success Stories
News
Events
Download the latest version for Windows
Download Python 3.9.0
Looking for Python with a different OS? Python for Windows, Linux/UNIX, Mac OS X, Other
Want to help test development versions of Python? Prereleases, Docker images
Looking for Python 2.7? See below for specific releases

Figure 1.1 Python Website: Download the Latest Version

In this Study Guide, we will standardise the operating system to Windows 10. Users of Linux/UNIX, Mac OS X or other operating systems can find equivalent applications to execute the same steps. After downloading the installer and double-clicking on it, the following window will then appear:


Python 3.9.0 (64-bit) Setup

Install Python 3.9.0 (64-bit)

Select Install Now to install Python with default settings, or choose Customize to enable or disable features.

Install Now
C:\Users\Karl\AppData\Local\Programs\Python\Python39

Includes IDLE, pip and documentation
Creates shortcuts and file associations

Customize installation
Choose location and features

☑ Install launcher for all users (recommended)
☑ Add Python 3.9 to PATH

Figure 1.2 Python Installation on Windows: Check the Box to Add Python to PATH

Note that when installing Python on Windows, ensure that the box "Add Python [Version] to PATH" as in Figure 1.2 is checked.

Furthermore, we will need Atom as our editor for composing Python scripts. Though we can also execute Python codes without writing them in an editor, it is much more convenient to do so. We can download Atom text editor at https://atom.io:

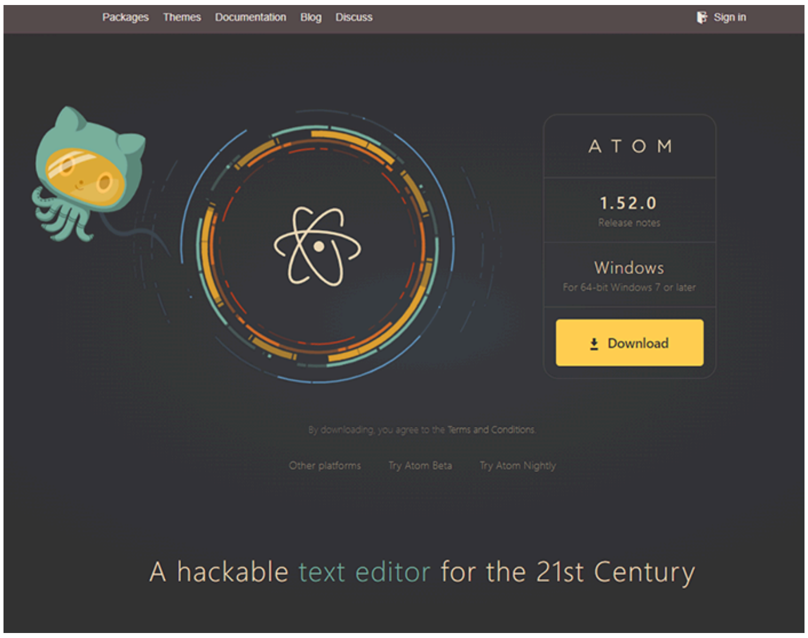


Figure 1.3 Download Website of Atom for Windows

We can select the operating system that we prefer for the Atom installer. Press the "Download" button and install Atom by executing the installer after the download has completed.

Read

Read the following two sections of the textbook on installing Python 3 on mac OS or Windows:

Exercise 0. The Setup (Windows)

Exercise 0. The Setup (maxOS)

1.2 Writing and Executing Python Programs

After the installations, we can start writing our Python program. One simple way is to write and run in Python directly, which we can find in the start menu:

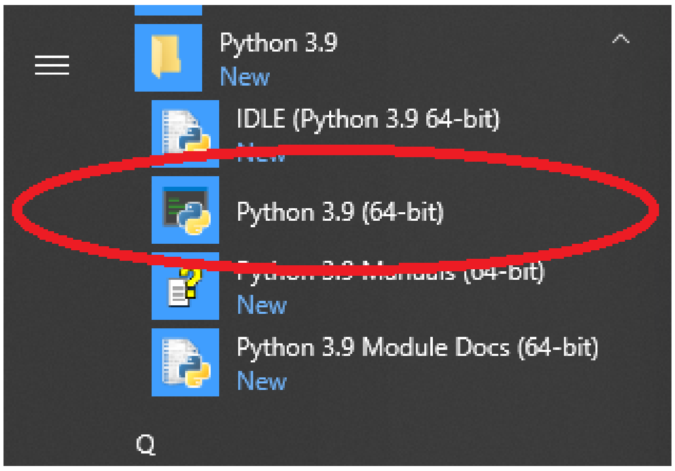
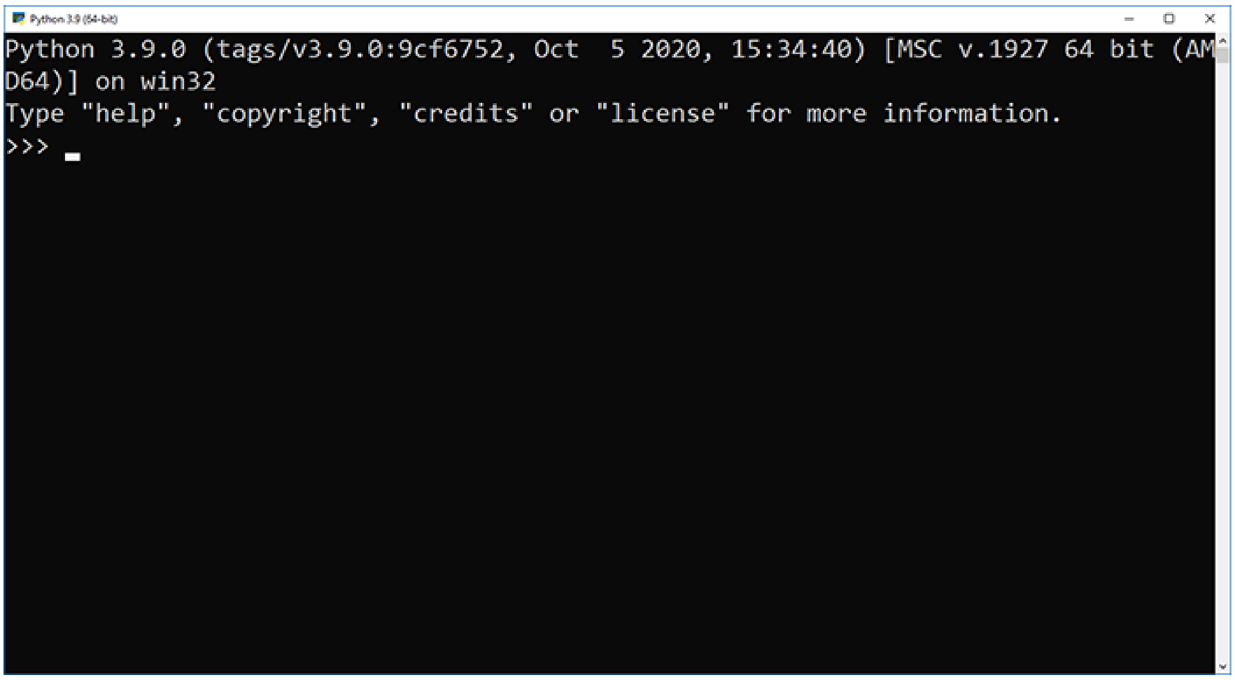


Figure 1.4 Finding Python in the Start Menu



```
Python 3.9.0 (tags/v3.9.0:9cf6752, Oct  5 2020, 15:34:40) [MSC v.1927 64 bit (AM
D64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> _
```

Figure 1.5 Python Interpreter

Once we see the “>>>” prompt, we can type in our Python command and let the Python interpreter execute it by pressing ENTER once the syntax is completed.

Another way to run Python is to call it from a terminal app. For this, we will need Windows PowerShell or Command Prompt to open it. Type “PowerShell” or “Command Prompt” in the “Search Windows” box on the task bar:

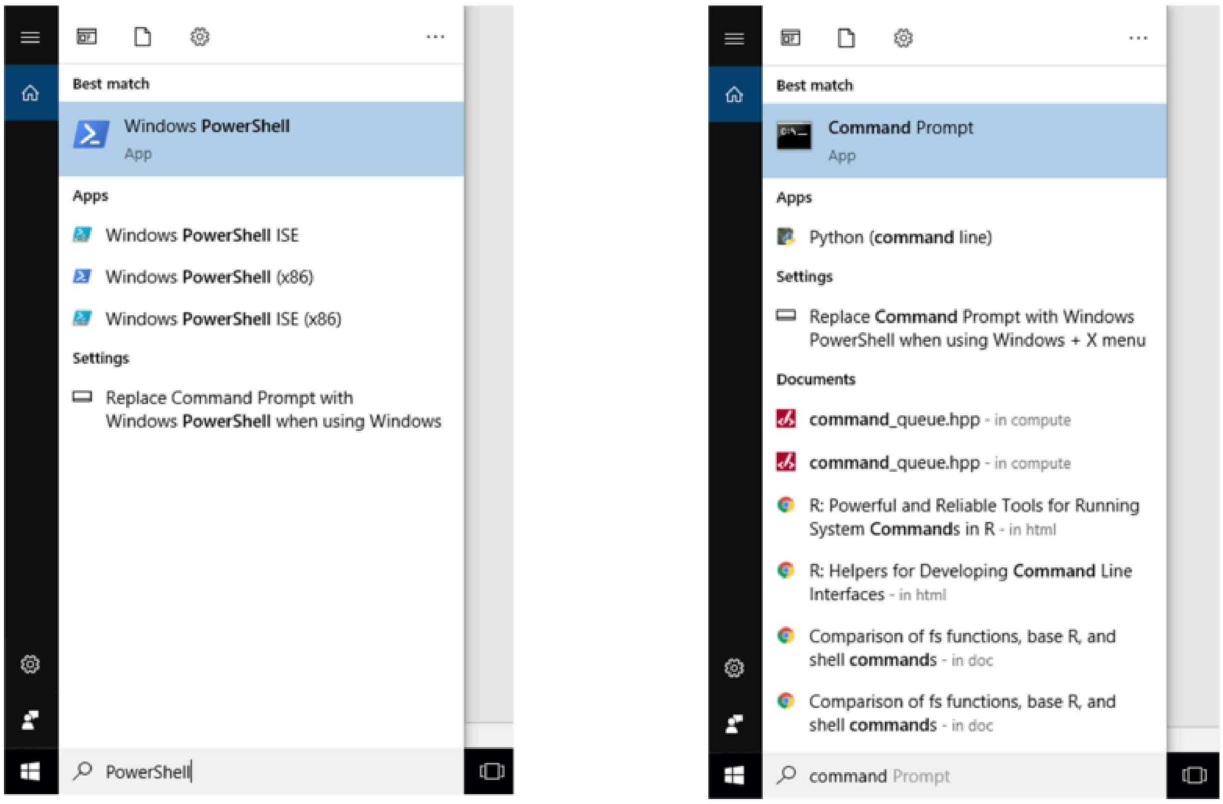
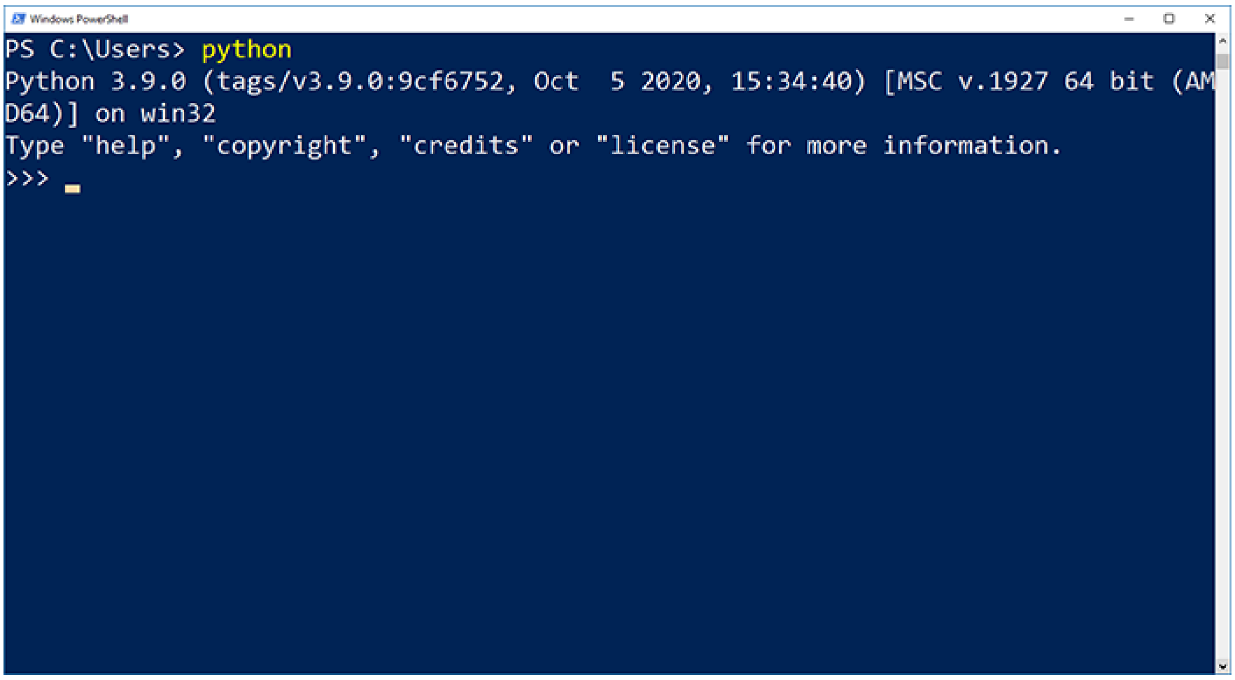

Best match
Windows PowerShell
App
Apps
Windows PowerShell ISE
Windows PowerShell (x86)
Windows PowerShell ISE (x86)
Settings
Replace Command Prompt with Windows PowerShell when using Windows
PowerShell
Best match
Command Prompt
App
Apps
Python (command line)
Settings
Replace Command Prompt with Windows PowerShell when using Windows + X menu
Documents
command_queue.hpp - in compute
command_queue.hpp - in compute
R: Powerful and Reliable Tools for Running System Commands in R - in html
R: Helpers for Developing Command Line Interfaces - in html
Comparison of fs functions, base R, and shell commands - in doc
Comparison of fs functions, base R, and shell commands - in doc
command Prompt

Figure 1.6 Searching for Windows PowerShell and Command Prompt

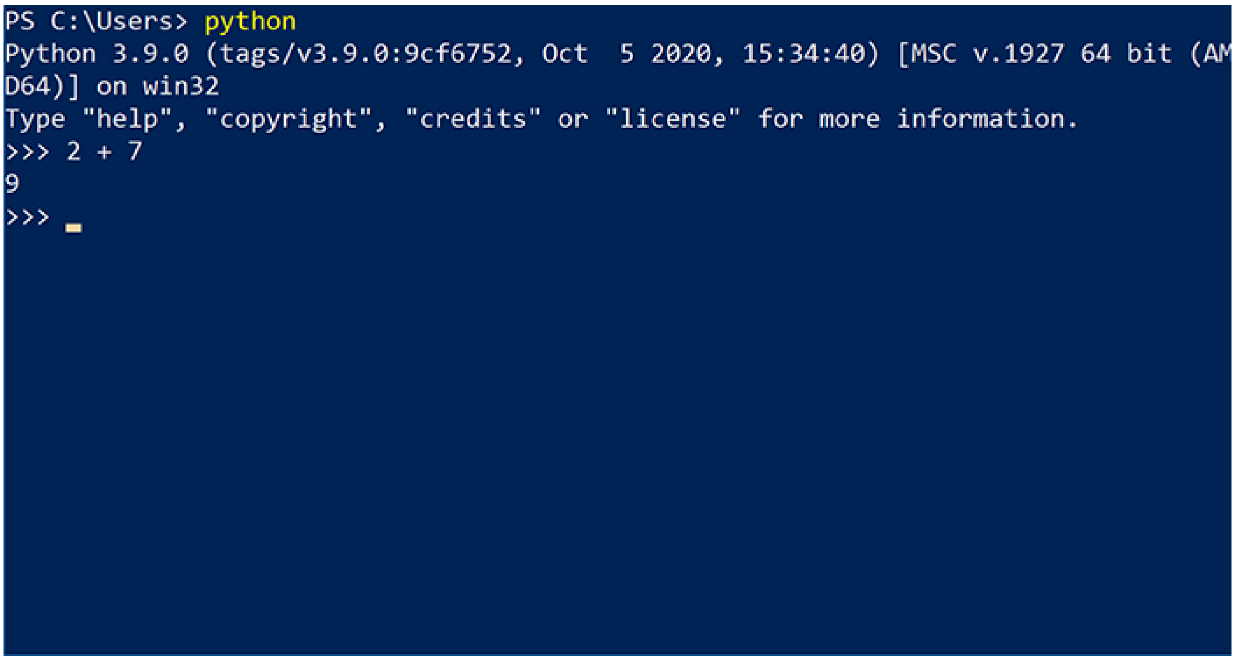
For simplicity, we will use Windows PowerShell in the following. Type "python" in PowerShell and then press ENTER:



```
PS C:\Users> python
Python 3.9.0 (tags/v3.9.0:9cf6752, Oct  5 2020, 15:34:40) [MSC v.1927 64 bit (AM
D64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> _
```

Figure 1.7 Launching Python in Windows PowerShell

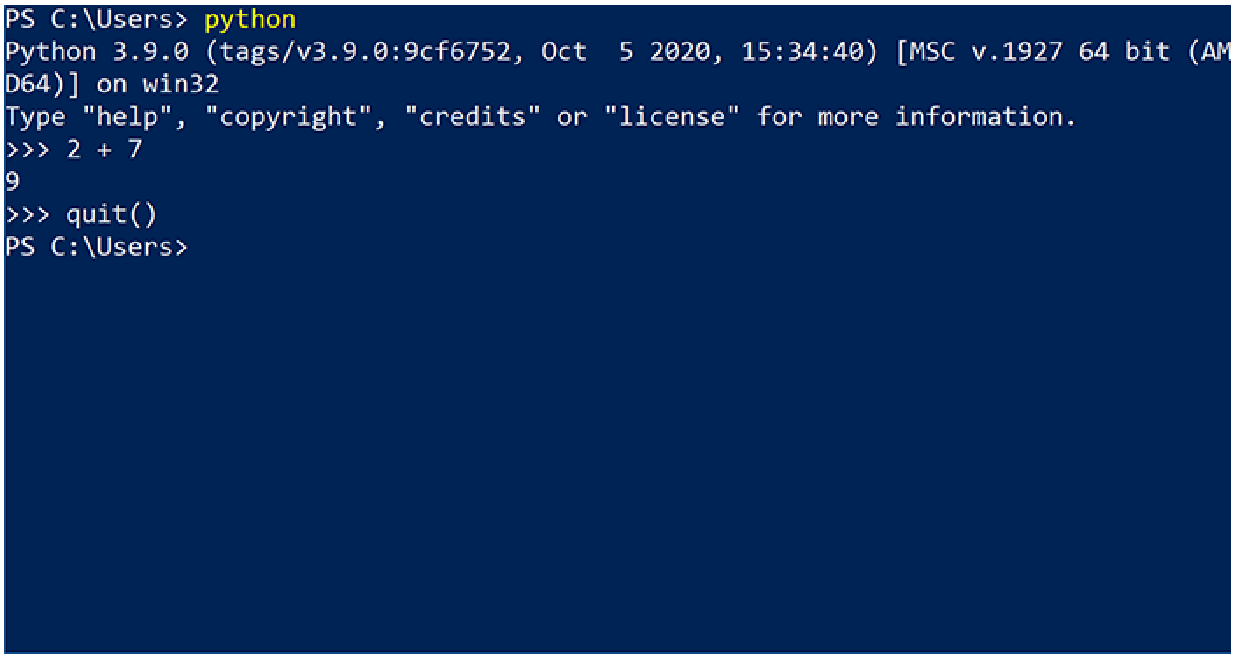
Once the Python interpreter has been started, we can see a very similar screen layout as shown in Figure 1.5. Now we can type in our Python code and immediately see the output once we have pressed ENTER. For instance, if we would like to do a simple calculation such as `2 + 7`, we simply type in this equation and press ENTER. Python will interpret what we have typed in and print the result in the next line.



```
PS C:\Users> python
Python 3.9.0 (tags/v3.9.0:9cf6752, Oct  5 2020, 15:34:40) [MSC v.1927 64 bit (AM
D64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> 2 + 7
9
>>> _
```

Figure 1.8 Execute a Simple Python Code

After we have finished executing our Python programs, we can quit Python by entering `quit()` and then press ENTER. We will return to the prompt of the operating system same as what we had seen before we started Python.



```
PS C:\Users> python
Python 3.9.0 (tags/v3.9.0:9cf6752, Oct  5 2020, 15:34:40) [MSC v.1927 64 bit (AM
D64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> 2 + 7
9
>>> quit()
PS C:\Users>
```

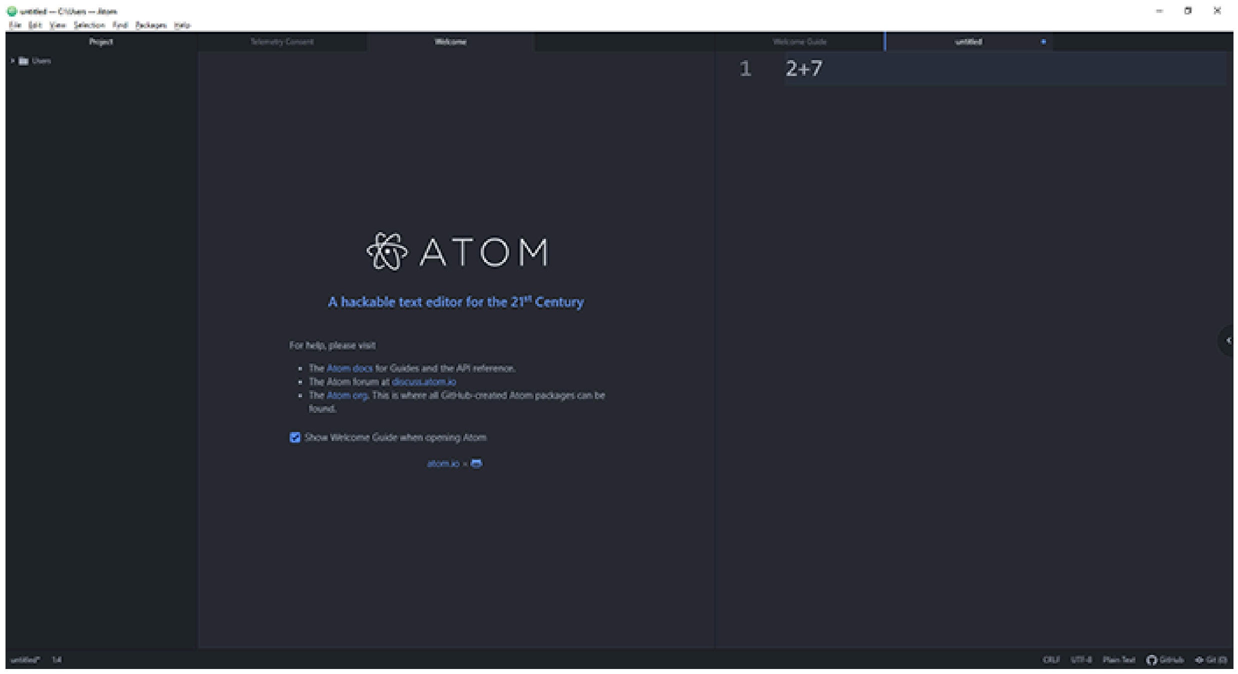
Figure 1.9 Quit the Python Interpreter

Apart from interactively working with the Python interpreter, we can also let Python run our own program scripts. These scripts are text files saved with the extension .py. In these script files, we put all the Python codes to be executed in a batch, instead of typing in and executing the syntax line by line like in the interpreter.

One obvious advantage of using a script file is that in most of the cases, we may intend to do a couple of calculations and data manipulation steps before asking Python to return the final output to us. Some of these executions could be quite inconvenient, or perhaps even impossible, if we must run them line by line.

We will use Atom as the editor to compose our Python scripts in the first part of this study guide, and then execute these scripts using the python command in PowerShell.

Now, we open Atom and write our first program.

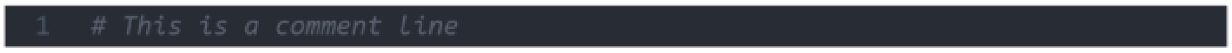

ATOM
A hackable text editor for the 21st Century
For help, please visit
The Atom docs for Guides and the API reference.
The Atom forum at discuss.atom.io
The Atom org. This is where all GitHub-created Atom packages can be found.
Show Welcome Guide when opening Atom
atom.io

```
2+7
```

Figure 1.10 Writing Python Code with Atom

Another advantage of writing Python code in a script is that we can add comments to it. Comments are lines in a script that will not be executed by Python. We can use comments to explain the procedures that the Python code is executing. Including comments is important since they will make it more readable and understandable for future editing or debugging, and simplify the overall maintenance of the program.

Comments in Python scripts start with a hash (#). After the hash, we can type in our explanations or descriptions of the referred syntaxes. Comments can be placed as a complete single line or after a line of syntax. If we need multiple lines for our comments, we will have to start every comment line with a hash.



```
# This is a comment line
```

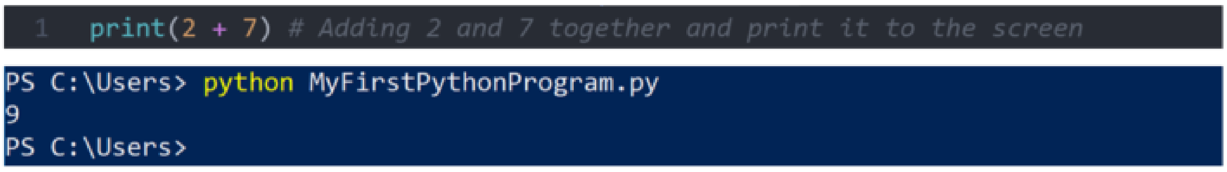
Figure 1.11 A Comment Line in Python

After editing the Python commands in a script using Atom, we can save the script as a .py text file. Different from the Python interpreter, we need to use the `print()` function explicitly to generate an output to the screen while Python is executing the script.

In the PowerShell, we need to change to the directory where we have saved the Python script and then run the script by executing the following command:

```
python filename.py
```

Note that `filename` is the file name of our Python script.



```
print(2 + 7) # Adding 2 and 7 together and print it to the screen
```

```
PS C:\Users> python MyFirstPythonProgram.py
9
PS C:\Users>
```

Figure 1.12 Executing a Python Scripts written in Atom Using PowerShell

Figure 1.12 shows us how PowerShell presents the output of a Python script. Since our program asks Python to print out the result of the addition `2 + 7`, the Python interpreter will execute the arithmetic operation in the background and return the value to the function `print()` for output. After Python has executed the whole script, it will return to the prompt of the operating system.

Read

Read the following section of the textbook on examples of composing and executing Python scripts:

Exercise 1 A Good First Program (Windows)

Exercise 1 A Good First Program (macOS)

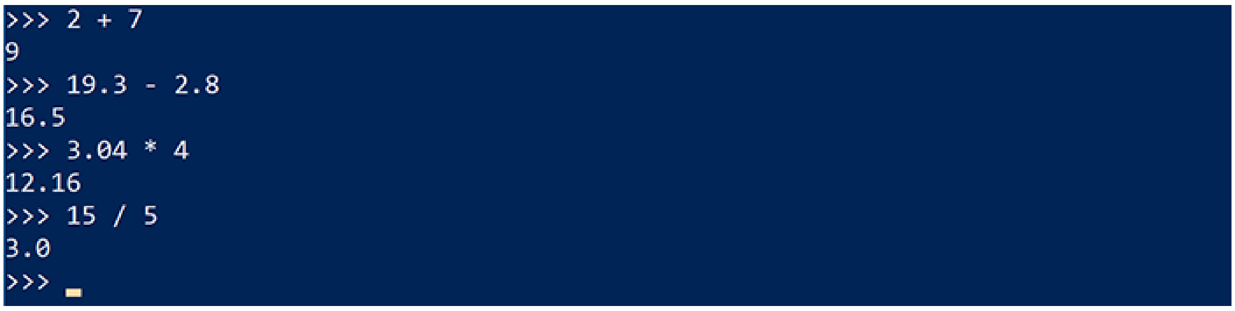
Chapter 2: Basic Arithmetic and Variables

Lesson Recording

Basic Arithmetic and Variables in Python

2.1 Arithmetic Operators

Before we start writing more sophisticated programs, we shall first go one step backwards and familiarise ourselves with the most origin function of a computer: calculation. Python can be powerful in many ways, but we can also use it for very trivial tasks such as adding two numbers together. In Figure 1.8 and Figure 1.12, we instruct Python to carry out a simple addition 2 + 7 for us. Similarly, we can also command Python to do other basic arithmetic operations.



```
>>> 2 + 7
9
>>> 19.3 - 2.8
16.5
>>> 3.04 * 4
12.16
>>> 15 / 5
3.0
>>> 
```

Figure 1.13 Simple Calculations with Python

The following Python arithmetic operators are available for mathematical calculations:

Table 1.1 Python Arithmetic Operators

Operator	Function	Example
+ Addition	Adds values on either side of the operator	`10 + 20 = 30`
- Subtraction	Subtracts right-hand operand from left-hand operand	`10 – 20 = -10`
* Multiplication	Multiplies values on either side of the operator	`10 * 20 = 200`
/ Division	Divides left-hand operand by right-hand operand	`20 / 10 = 2`
% Modulus	Divides left-hand operand by right-hand operand and returns remainder	`20 % 10 = 0`
** Exponent	Performs exponential (power) calculation on operators	`10 ** 20 = 10,000,000,000,000, 000,000`

// Floor Division	The division of operands where the result is the quotient in which the digits	`9 // 2 = 4`
		`9.0 // 2.0 = 4.0`
		`-11 // 3 = -4`

Operator	Function	Example
	after the decimal point are removed. But if one of the operands is negative, the result is floored, i.e., rounded away from zero (towards negative infinity)	`-11.0 // 3 = -4.0`

(Source: https://www.tutorialspoint.com/python/python_basic_operators.htm)

Same as normal mathematics, the exponent operator has higher priority than the operators of multiplication or division, which in turn will be calculated prior to the addition or subtraction operators. Furthermore, we can add parentheses to our equation to indicate that those terms within a parenthesis should have the highest priority in the calculation.

Note that mathematical functions such as the square root, the logarithm, the exponential, or the trigonometrical functions are not included in basic Python. If we want to include these functions in our calculation, we will need to import packages such as “math” or “NumPy” in our code. We will discuss how to import and call functions from external libraries or packages in Study Unit 2.

And then there are other operators in Basic Python such as relational operators, logical operators, etc. We will also discuss them at a later stage of this study unit.

Read

Read https://www.tutorialspoint.com/python/python_basic_operators.htm for more about basic operators in Python.

2.2 Variables

In most of the situations, we wish to write programs that help us automate routine operations without adjusting our programs according to the actual needs. For instance, we may not always want to add 2 and 7 together. Instead, we would prefer to let the computer add up any pair of arbitrary numbers for us, and we can choose these numbers depending on the situation. As a result, we would like to keep our program as general as possible by using variables instead.

In python, we define a variable by its name which is an arbitrary combination of characters (A-Z, a-z), underscores (_) and numbers (0-9). Subsequently, we assign a value to the variable and let Python operate with it. And we can change the value of the defined variable at any stage of our program.

To assign a value to a variable, all we need to do is:

```
variable = value
```

Remember that it is important to put the variable left of the equal sign (=) and the value right of it. If we switched their positions, it would be equivalent to assigning a name to a number. This would result in a syntax error, and Python will stop executing the rest of the program at once.

In Python, the name of a variable can be short (e.g., `x`, `y`, `z`) or more descriptive (e.g., `age`, `carname`, `total_volume`). But there are certain rules which we must follow when we create our variable names.

- A variable name must start with a letter or an underscore (_).
- A variable name cannot start with a number.
- A variable name can only contain alpha-numeric characters and underscores (A-z, 0-9, and _).
- Variable names are case-sensitive (age, Age and AGE are three different variables).

(Source: https://www.w3schools.com/python/python_variables.asp)

Here are some examples of valid variable names:

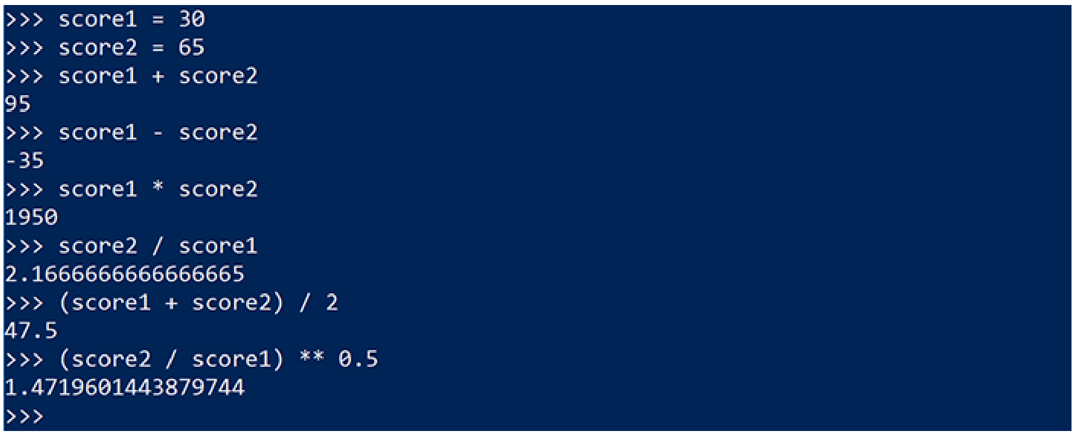
```
myvar = 10

my_var = 10

my_var = 10

myVar = 10

MYVAR = 10

_myvar2 = 10
```

Here are some examples of invalid variable names:

```
2myvar = 10

my-var = 10

my var = 10
```

Once values are assigned to variables, we can use them for any arithmetic operations as introduced in Chapter 2.1 for numeric values.

Example (Students' score): Suppose we have the exam scores of two students, 30 and 65, and we would like to store them in two variables, `score1` and `score2`, for some mathematical operations. Subsequently, we can conduct arithmetic operations with these variables.



```
>>> score1 = 30
>>> score2 = 65
>>> score1 + score2
95
>>> score1 - score2
-35
>>> score1 * score2
1950
>>> score2 / score1
2.1666666666666665
>>> (score1 + score2) / 2
47.5
>>> (score2 / score1) ** 0.5
1.4719601443879744
>>>
```

Figure 1.14 Assigning Values to Variables for Operations

Read

Read the following section of the textbook on examples of creating and using variables in Python:

Exercise 4 Variables and Names

2.3 Types of Variable and Expressions

In Python, there are different types of variable that we can work with. In the previous section, we assign numeric values to variables which makes them numeric variables. Nevertheless, there are also different types of numeric variable. Here are some main types of variable available in Python:

Table 1.2 Types of Variable in Python

Type	Description	Example
Integer	The value of an integer variable must be an integer, a value without decimal point. It can be both positive and negative.	`a = 5`
Float	The value of a float variable can be an arbitrary numeric value with a floating point.	`b = 10.5`

String	The value of a string variable can contain any letters in both cases, special characters as well as numbers. Note that if numbers are assigned to a string variable, no mathematical operations can be applied on it.
	
	To assign a value to a string variable, the value must be written between a pair of quotation marks (it can be single or double quotation mark, but it must be consistent for the same value).
		`c = "John"`
		
		`d = "Tan"`

	Furthermore, two strings can be concatenated by being "added up".	`c + d = "JohnTan"`
Boolean (Bool)	The value of a Boolean variable can be either `True` or `False`.	`e = True`

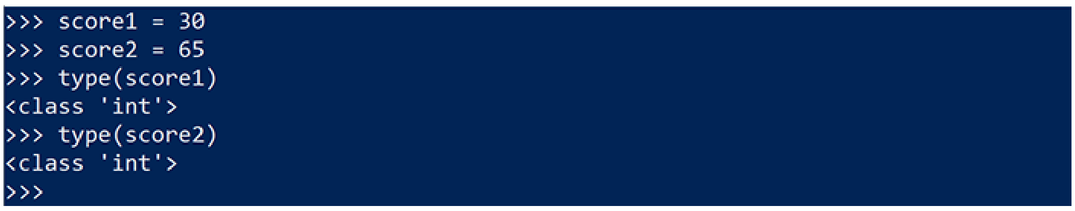
In the following, we will use the general term *expression* for variables or when they are linked with operators. For instance, `a + b` is an expression and not a variable, unless we define `c = a + b` in our program where `c` is then a new variable. However, we would rather refer to expressions in our program directly since we do not always define new variables for calculation steps in between.

To check the type of a variable, we can use the `type()` function on any variable in our program.

```
type(variable_name)
```

Python will then print the variable type such as “int” (for integer), “float” (for float), or “str” (for character string) to the screen.

Example (Cont’d): In Figure 1.14, we assign 30 and 65 to the two variables, `score1` and `score2`, respectively. We can use the `type()` function to check their variable type.



```
>>> score1 = 30
>>> score2 = 65
>>> type(score1)
<class 'int'>
>>> type(score2)
<class 'int'>
>>>
```

Figure 1.15 Checking Variable Type

We can see that Python returns `<class  'int'>` as the screen output. The information we are enquiring is put in the single quotation marks like `int` in this case. As a result, we can see that both `score1` and `score2` are integer variables.

Read

Read the Python documentation (https://docs.python.org/3/library/stdtypes.html#numeric-types-int-float-complex) for more about the different operations on numeric data types.

Chapter 3: Print and Input

Lesson Recording

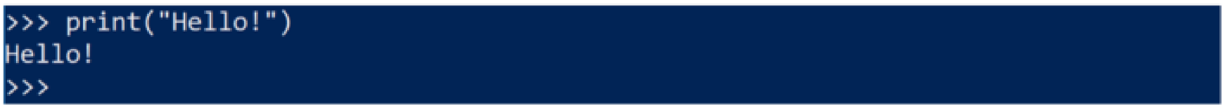
Print and Input in Python

3.1 Printing

Writing programs is not only to automate routine operations by the computer. It is also of interest to show the results, information, or messages to the user while the Python program is running. We can use the `print()` function to generate screen output for the user to read.

```
print("My String")
```

All we need to do here is to put the text within a pair of quotation marks and pack everything inside the `print()` function. While the program is being executed, Python will then extract the content within the quotation marks and print it onto the screen.

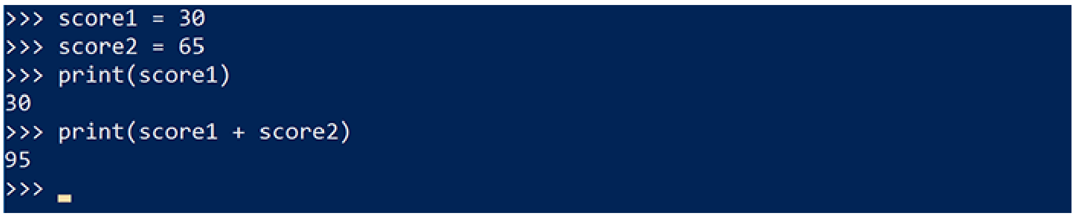


```
>>> print("Hello!")
Hello!
>>>
```

Figure 1.16 Simple Use of the print() Function

The print function is not only limited to print pre-defined strings. We often wish to print out the result of a calculation, as shown in Figure 1.8 and Figure 1.12, or the value of a variable, or the result of a calculation based on variables as well.

Example (Cont'd): Suppose we would like Python to print the exam score of student1, `score1`, and the sum of the two scores, `score1 + score2`, onto the screen.



```
>>> score1 = 30
>>> score2 = 65
>>> print(score1)
30
>>> print(score1 + score2)
95
>>>
```

Figure 1.17 Printing Variables and Their Calculations

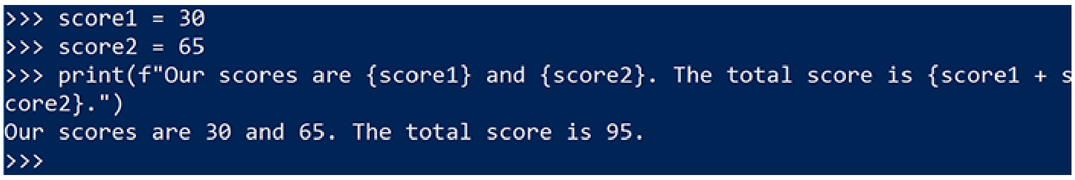
3.2 String Formatting

In the last command of Figure 1.17, we ask Python to first execute the mathematical operation `score1 + score2`, and then print it out to the user. Basically, there is nothing wrong with this screen output, it only looks a bit "cold" and rather not interactive. To make such an output to look more like a statement addressing to the user, we can mix a statement with variables or expressions in our printing string to become a formatted string for printing.

```
print(f"My String {expression1} {expression2} …")
```

The `print`-command is almost identical to the one for normal printing. The only difference here is to put an "`f`" before the open quotation mark. Subsequently, we can place the variables or expressions that we would like to print anywhere within the text and wrap it within a pair of curly brackets `{}`.

Example (Cont'd): Suppose we would like to print the scores of the two students as well as their sum in a sentence such as "Our scores are 30 and 65. The total score is 95". We can use the following code to create this screen output.

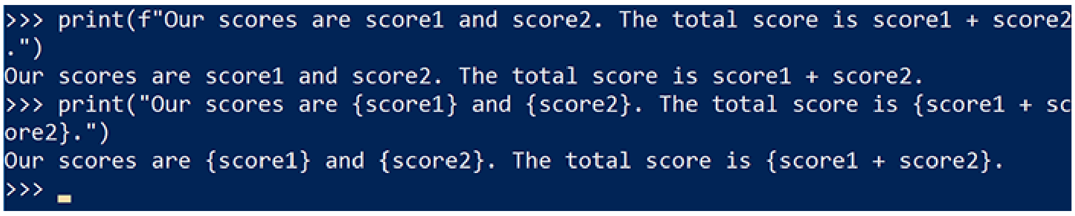


```
>>> score1 = 30
>>> score2 = 65
>>> print(f"Our scores are {score1} and {score2}. The total score is {score1 + s
core2}.")
Our scores are 30 and 65. The total score is 95.
>>>
```

Figure 1.18 Printing Formatted String

Unlike the entire argument within the quotation marks, every expression written inside the curly brackets of a formatted string will be evaluated before being printed onto the screen. In Figure 1.18, the expressions `score1`, `score2` and `score1 + score2` will be evaluated first. That is, Python will execute the `print()` function with the value assigned to these expressions and not with the expressions as part of the string.

Example (Cont'd): If the variables or expressions were not put inside some curly brackets as a formatted string, the expressions `score1`, `score2`, and `score1 + score2` would be treated as ordinary strings and printed just as how they were written. And if the "`f`" were forgotten at the beginning of the argument in the `print()` function, Python would interpret the missing of the "`f`" as an instruction to print the entire text within the quotation mark without evaluating the expressions in the curly brackets first. Since the curly brackets are part of the string in the `print()` function, they will be printed as well.



```
>>> print(f"Our scores are score1 and score2. The total score is score1 + score2
.")
Our scores are score1 and score2. The total score is score1 + score2.
>>> print("Our scores are {score1} and {score2}. The total score is {score1 + sc
ore2}.")
Our scores are {score1} and {score2}. The total score is {score1 + score2}.
>>>
```

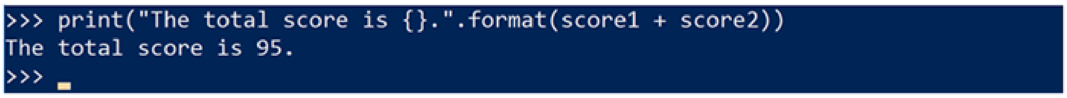
Figure 1.19 Output of Incomplete String Formatting Syntax

We can also use the `.format()` method for string formatting. Note that the `.format()` method only takes one expression in its argument.

```
print("My String {}".format(expression))
```

For the `.format()` method, we need to place the curly brackets at the position within the string where we would like to print our expression.

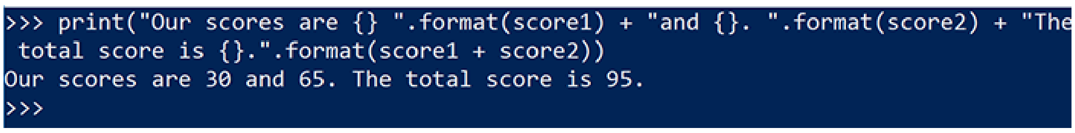
Example (Cont'd): We can print the total score of the two students at the end of a statement such as "The total score is 95." by using the `.format()` method.



```
>>> print("The total score is {}.".format(score1 + score2))
The total score is 95.
>>> 
```

Figure 1.20 Simple Usage of the `.format()` Method

Figure 1.21 shows us how the printing string can be extended in order to get the same output as in Figure 1.18. But the syntax is much longer here.



```
>>> print("Our scores are {} ".format(score1) + "and {}. ".format(score2) + "The
 total score is {}.".format(score1 + score2))
Our scores are 30 and 65. The total score is 95.
>>>
```

Figure 1.21 Multiple Usage of `.format()` Method

Nevertheless, the `.format()` method can also be useful if we have one variable to be printed at the end of our statement.

Read

Read the following section of the textbook on printing formatted strings:

Exercise 5 More Variables and Printing

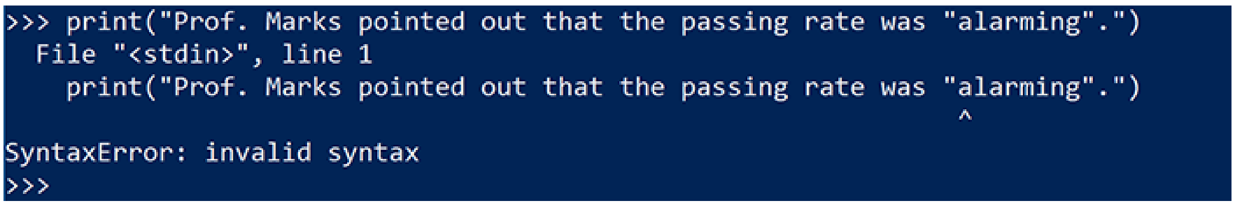
Read the following two sections of the textbook on printing formatted strings using the `.format()` method:

Exercise 6 Strings and Text

Exercise 7 More Printing

3.3 Escape Sequences

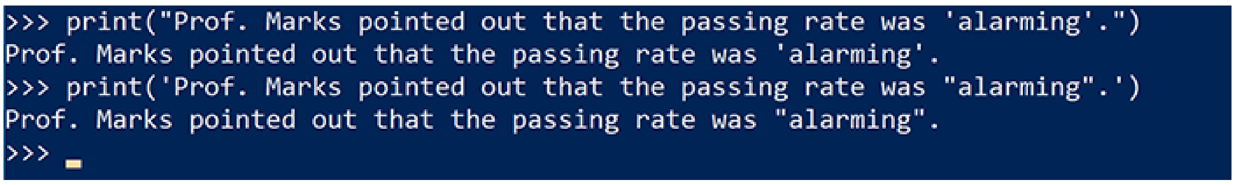
Escape sequences are used to print special characters that are invisible such as ENTER, or characters that can cause syntax error such as single (`'`) or double quotation marks (`"`). Suppose we would like to include a quote within a string for the screen output.



```
>>> print("Prof. Marks pointed out that the passing rate was "alarming".")
  File "<stdin>", line 1
    print("Prof. Marks pointed out that the passing rate was "alarming".")
                                                                       ^
SyntaxError: invalid syntax
>>>
```

Figure 1.22 Syntax Error Caused by Quotation Marks within a string

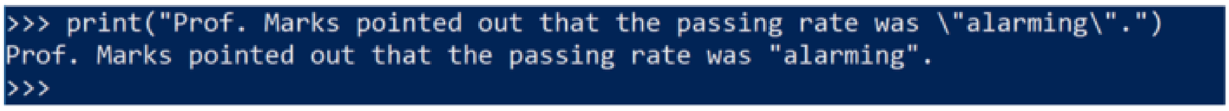
In Figure 1.22, the string in the `print()` function ends with the second quotation mark. Everything subsequent to it will be interpreted as part of the code. Since the word "alarming" is neither a Python command nor a variable, Python simply interprets it as an erroneous syntax. One way to avoid this error is to use single quotation marks for either the citation quote or the string definition.



```
>>> print("Prof. Marks pointed out that the passing rate was 'alarming'.")
Prof. Marks pointed out that the passing rate was 'alarming'.
>>> print('Prof. Marks pointed out that the passing rate was "alarming".')
Prof. Marks pointed out that the passing rate was "alarming".
>>> _
```

Figure 1.23 Printing Quotation Marks within a String

Another way is to use the escape sequence `\"` within the string instead of switching between single and double quotation marks.

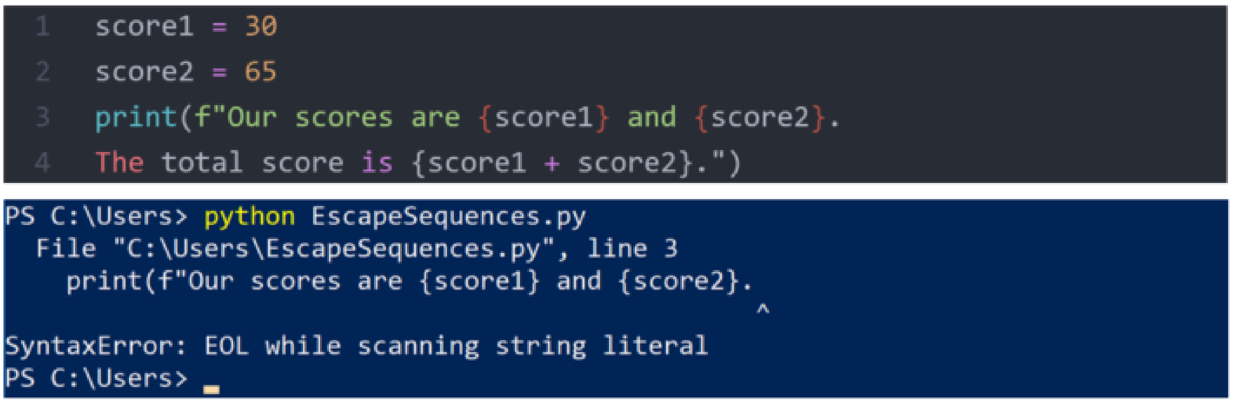


```
>>> print("Prof. Marks pointed out that the passing rate was \"alarming\".")
Prof. Marks pointed out that the passing rate was "alarming".
>>>
```

Figure 1.24 Printing Quotation Marks using Escape Sequence

Escape sequences are also useful when line breaks should be inserted within a string. By adding the escape sequence “`\n`” at the position within the string, the subsequent part of the string will be printed in the next line of the output screen.

Example (Cont’d): Now we would like to print the first and the second sentences in Figure 1.18 in two separate lines. However, we would create a syntax error if we just placed a line break in our Python script.

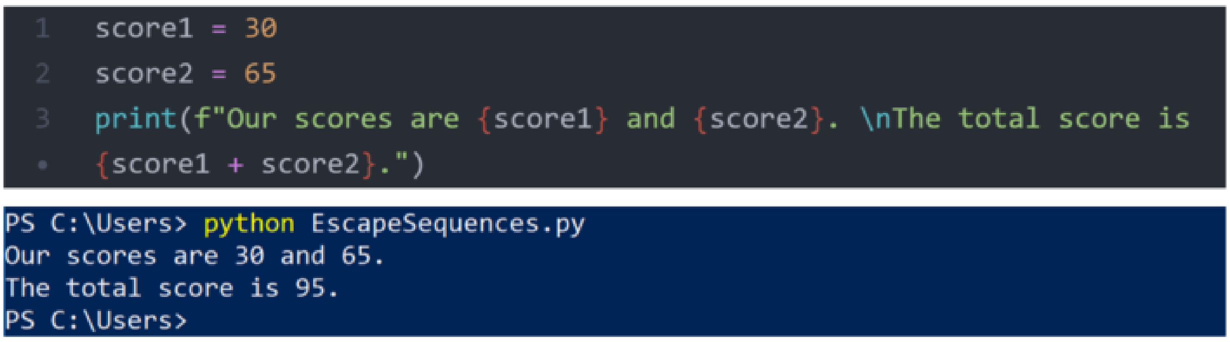


```
score1 = 30
score2 = 65
print(f"Our scores are {score1} and {score2}.
The total score is {score1 + score2}.")
```

```
PS C:\Users> python EscapeSequences.py
  File "C:\Users\EscapeSequences.py", line 3
    print(f"Our scores are {score1} and {score2}.
                                                 ^
SyntaxError: EOL while scanning string literal
PS C:\Users> _
```

Figure 1.25 Erroneous Line Breaks within a String

Figure 1.25 illustrates that Python treats such a line break within a string as a syntax error. The reason is that the string in the `print()` function must be closed by a quotation mark in the same line. Instead of closing the first line directly and start a new `print()` function in the second line to solve this problem clumsily, we can add an escape sequence `\n` into the string:



```python
score1 = 30
score2 = 65
print(f"Our scores are {score1} and {score2}. \nThe total score is
{score1 + score2}.")
```

```
PS C:\Users> python EscapeSequences.py
Our scores are 30 and 65.
The total score is 95.
PS C:\Users>
```

Figure 1.26 Line Breaks Created by Escape Sequence `\n`

The following list contains some useful escape sequences available in Python:

Table 1.3 List of Some Escape Sequences Available in Python

Escape Sequences	Description	Example

`\newline`	Backslash and newline ignored	`>>> print("line1 \`
		`Line2")`
		`line1 line2`

`\\`	Backslash (\)	`>>> print("\\")`
		`\`

`\'`	Single quote (')	`>>> print("\'")`
		`'`

`\"`	Double quote (")	`>>> print("\"")`

Escape Sequences	Description	Example
		`"`

`\n`	ENTER or line break	`>>> print("line1 \n line2")`
		`line1`
		`line2`

`\b`	Backspace (BS)	`>>> print("line1 \b line2")`
		`line1 line2`

`\t`	Horizontal Tab (TAB)	`>>> print("line1 \t line2")`
		`line1 line2`

`\v`	Vertical Tab (VT)	`>>> print("line1 \vline2")`
		`line1`
		`line2`

(Source: https://www.python-ds.com/python-3-escape-sequences)

Read

Read the following two sections of the textbook on using escape sequence in printing formatted strings as in Figure 1.8:

Exercise 9 Printing, Printing, Printing

Exercise 10 What Was That?

3.4 Input

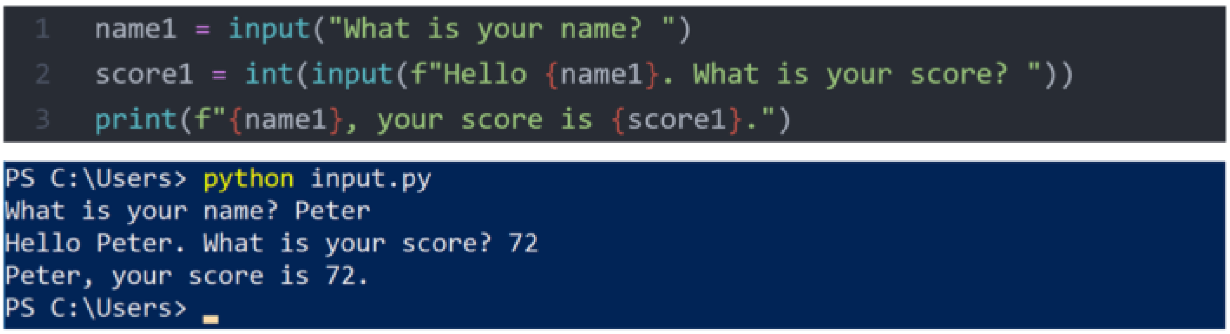
While a program script is being executed, it requires values to be assigned to the variables in order to proceed in its instructions. So far, we have discussed the possibility to assign values to the variables in the script. That means, the values are fixed when the program started to run. However, in most of the cases, those values are unknown and can only be assigned while the program is running, mostly based on the input of the user. In Python, we can use the `input()` function to ask the user to enter the value for a variable.

```
variable = input("My String")
```

The whole syntax will be put on the right-hand side of an equal sign so that Python can assign the user input to the variable that is defined on the left-hand side of the same equal sign.

Unlike the `print()` function, Python requires the user to type in something and then press ENTER to complete the execution of the `input()` function. Same as the `print()` function, we can instruct Python to print a string to the screen within the `input()` function. Usually, this string should be a question and/or some instructions to inform the user what they shall input here. Furthermore, we are also allowed to mix the assigned values of some variables with the instruction text to become a formatted string that will be printed on the screen for the subsequent input.

Example (Cont'd): Instead of pre-assigning values to the variables, we will ask the user to enter his/her name and his/her score. Subsequently, we will print out his/her score by addressing his/her name and embed his/her score in a sentence such as "Your score is ...".



```python
name1 = input("What is your name? ")
score1 = int(input(f"Hello {name1}. What is your score? "))
print(f"{name1}, your score is {score1}.")
```

```
PS C:\Users> python input.py
What is your name? Peter
Hello Peter. What is your score? 72
Peter, your score is 72.
PS C:\Users> _
```

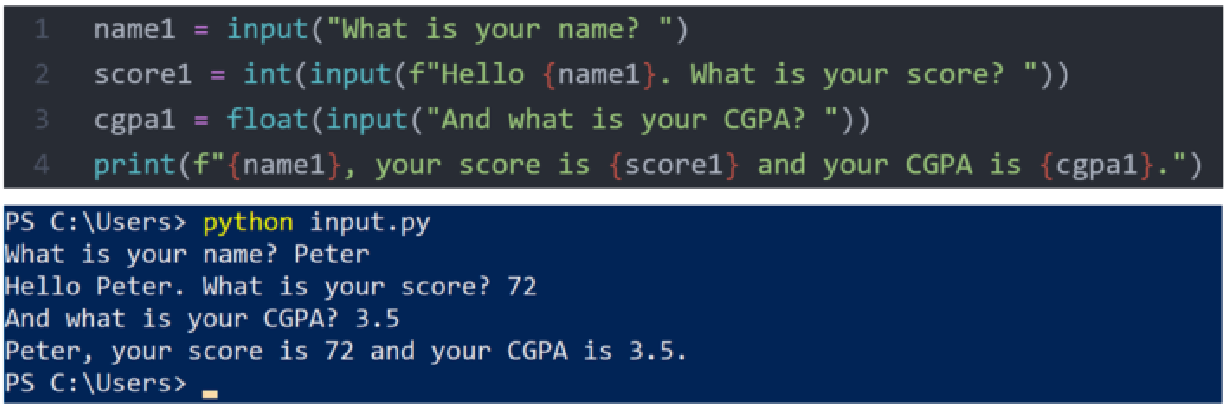
Figure 1.27 Example of Using `input()` Function

In Python, the value assigned by the user within an `input()` function will be stored as string. If the input should be an integer or a number with a floating point, we can convert the input using:

```
variable = int(input("My String"))
variable = float(input("My String"))
```

The functions `int()` and `float()` are used to convert a string variable to become an integer or a float variable. (Conversely, there is the `str()` function to convert an integer or a float variable into a string variable.)

Example (Cont'd): Since the score of a student must be an integer within 0 and 100, we can convert it to an integer by embedding the `input()` function within an `int()` function. At the same time, we add a new question to ask the student for his/her CGPA and convert it to a float variable.



```
name1 = input("What is your name? ")
score1 = int(input(f"Hello {name1}. What is your score? "))
cgpa1 = float(input("And what is your CGPA? "))
print(f"{name1}, your score is {score1} and your CGPA is {cgpa1}.")
```

```
PS C:\Users> python input.py
What is your name? Peter
Hello Peter. What is your score? 72
And what is your CGPA? 3.5
Peter, your score is 72 and your CGPA is 3.5.
PS C:\Users> _
```

Figure 1.28 Convert Input Value to Integer and Float

The syntax introduced above is to put the `input()` function inside the `int()` and `float()` functions and construct the instruction within a single line. Nevertheless, we can also separate these commands into two lines without changing the behaviour of the program:

```
variable = input("My String")
variable = int(variable)
```

These lines are certainly applicable to the `float()` function as well. It is noteworthy that if the user enters a value that is not a number, the `int()` or `float()` functions will interpret it as a syntax error and stop executing the code immediately. It is therefore a good programming habit to build in certain control mechanism for any user input command in our code. We will discuss the construction of such control mechanism in Chapter 5 of this study unit.

Read

Read the following two sections of the textbook on getting the user input:

Exercise 11 Asking Questions

Exercise 12 Prompting People

Chapter 4: If-elif-else-Conditions

Lesson Recording

If-elif-else-Conditions in Python

4.1 Boolean Expressions

To automate a routine by a computer program, we usually need to let the program "decide" what to execute in the next step based on some conditions. For instance, the user can choose to stay or quit the program after certain operations have been completed.

Before introducing the `if-elif-else`-conditions that Python uses to decide how the program should behave after a certain stage of the code, we need to get ourselves familiarised with the Boolean expressions first. As introduced in Chapter 2.3, Boolean variables have only two possible values: `True` and `False`. So, the basic concept of the conditional control flow is to evaluate whether a Boolean expression is `True` or not first, and then carry out either set of instructions depending on the evaluation.

A Boolean expression can be the result of a single relational operation or a combination of multiple relational operations linked by logical operators. Here are some relational operation examples:

Table 1.4 Examples of Relational Operation

Relational Operation	Result
`1 == 1`	`True`
`3 > 2`	`True`

Relational Operation	Result
`0 <= -5`	`False`
`a + b < 10`	`False if a + b >= 10`

In the above examples, the first two operations are obviously `True` since they correspond to the mathematical relationship between the left-hand and the right-hand sides of the equations. Note that if we want to check whether two expressions are identical, we will have to use the double equal sign (==) instead of the ordinary equal sign (=) since the single equal sign is used to assign a value to a variable. So, if we wrote `1 = 1` instead of `1 == 1`, a syntax error would return since Python would interpret our intention to be assigning a value to a number, which we know is not allowed from Chapter 2.2.

Below is a list of relational operators that we can use in Python.

Table 1.5 List of Relational Operators

Operator	Description	Example
`==`	`True` if the values of two operands are equal.	`(10 == 20)` is `False`.
`!=`	`True` if values of two operands are not equal.	`(10 != 20)` is `True`.
`<>`	`True` if values of two operands are not equal (Similar to the `!=` operator).	`(10 <> 20)` is `True`.
`>`	`True` if the value of left operand is greater than the value of right operand.	`(10 > 20)` is `False`.

Operator	Description	Example
`<`	`True` if the value of left operand is less than the value of right operand.	`(10 < 20)` is `True`.
`>=`	`True` if the value of left operand is greater than or equal to the value of right operand.	`(10 >= 20)` is `False`.
`<=`	`True` if the value of left operand is less than or equal to the value of right operand.	`(10 <= 20)` is `True`.

(Source: https://www.tutorialspoint.com/python/python_basic_operators.htm)

A Boolean expression can also be a combination of multiple relational operations, connected by the logical operators. Below is a list of logical operators in Python.

Table 1.6 List of Logical Operators

Operator	Description	Example
`and`	If both the operands are `True`, then the condition becomes `True`.	`(10 > 0 and 20 > 0)` is `True`.
`or`	If any of the two operands are non-zero, then the condition becomes `True`.	`(10 > 0 or 20 > 0)` is `True`.
`not`	Used to reverse the logical state of its operand.	`not(10 > 0 or 20 > 0)` is `False`.

(Source: https://www.tutorialspoint.com/python/python_basic_operators.htm)

The use of a single logical operation is usually quite straightforward as the operators are designed in a way that it just simply matches our spoken language. However, it could become quite confusing if we combine these operators in a Boolean expression. For instance, the following Boolean expressions are equivalent:

```
(a or b) and c        <==>     (a and c) or (b and c)

(a and b) or c        <==>     (a or c) and (b or c)

  not(a or b)         <==>      (not a) and (not b)

  not(a and b)        <==>       (not a) or (not b)
```

These are just a few examples and can be extended endlessly. It is utmost important to get familiarised on how to create Boolean expressions using relational and logical operators. Any failure in combining these operators could lead to unexpected behaviour of our program. The only, and most effective way here is to practise them with Python since we can check on the result directly.

Read

Read the following two sections of the textbook on Boolean expressions:

Exercise 27 Memorising Logic

Exercise 28 Boolean Practice

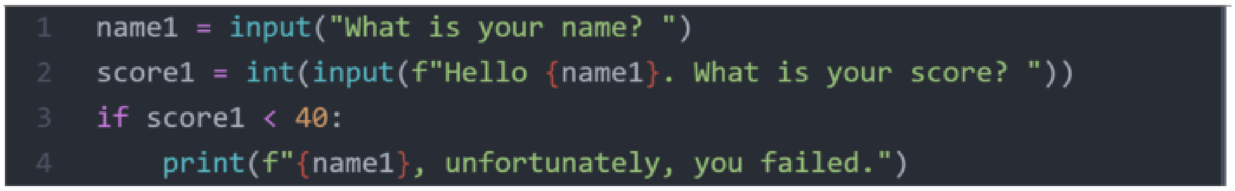
4.2 Conditional Statements

The result of a Boolean expression can serve as the condition that changes the behaviour of a program dynamically by embedding it in an `if`-conditional statement:

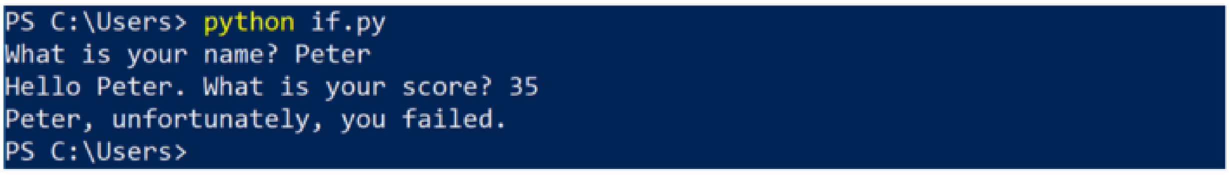
```
if condition:
     instructions
```

In the `if`-condition, Python will execute the syntaxes in the instructions if the condition is `True`. However, if the condition is `False`, Python will simply skip these lines and proceed with the subsequent code lines. Note that it is mandatory to put the colon directly behind the condition, and the instructions must be indented so that Python can interpret them as part of the `if`-block.

Example (Cont'd): If the score of a student is below 40, we will show a message on the screen to tell him/her that he/she failed in the exam.



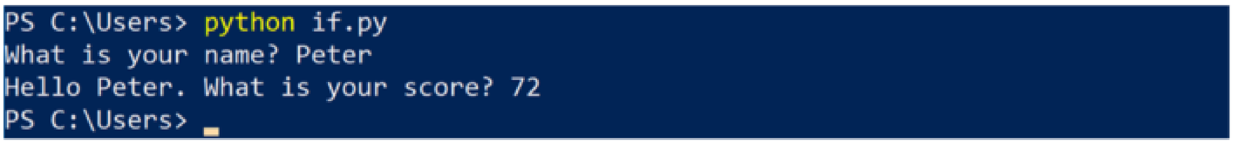
```python
name1 = input("What is your name? ")
score1 = int(input(f"Hello {name1}. What is your score? "))
if score1 < 40:
    print(f"{name1}, unfortunately, you failed.")
```



```
PS C:\Users> python if.py
What is your name? Peter
Hello Peter. What is your score? 35
Peter, unfortunately, you failed.
PS C:\Users>
```

Figure 1.29 `if`-Statement Example with `True` Condition

Figure 1.29 illustrates what Python does if the condition is `True`. On the other hand, if a student scores more than 40, nothing will be printed based on the Python script.



```
PS C:\Users> python if.py
What is your name? Peter
Hello Peter. What is your score? 72
PS C:\Users> _
```

Figure 1.30 `if`-Statement Example with `False` Condition

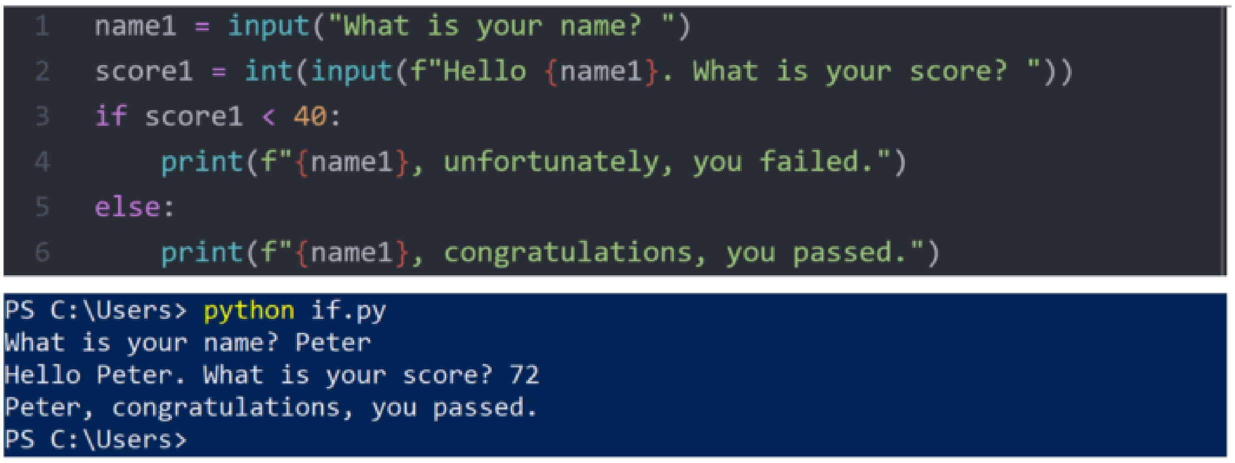
Figure 1.30 shows how Python skips all the instructions in the `if`-block since the condition is `False`.

If we intend to let Python execute another set of instructions if the condition is `False`, and not just skip the `if`-block, we can add an `else`-statement to the `if`-block:

```
if condition:
     instructions 1
else:
     instructions 2
```

Same as the `if`-condition, we must add a colon to the `else`-statement and the instructions following it must be indented as well.

Example (Cont'd): If the score of a student is below 40, we will show a message on the screen to tell him/her that he/she failed in the exam. Otherwise, we will show a message to tell him/her that he/she passed.



```python
name1 = input("What is your name? ")
score1 = int(input(f"Hello {name1}. What is your score? "))
if score1 < 40:
    print(f"{name1}, unfortunately, you failed.")
else:
    print(f"{name1}, congratulations, you passed.")
```

```
PS C:\Users> python if.py
What is your name? Peter
Hello Peter. What is your score? 72
Peter, congratulations, you passed.
PS C:\Users>
```

Figure 1.31 `if-else`-Statement Example

Figure 1.31 shows that if the condition is `False`, Python will execute those instructions following the `else`-statement.

If the construction of the condition allows more than two outcomes, we may need a third or fourth `if`-blocks, etc. In this case, we can use the `if`-`elif`-`else`-block:

```
if condition 1:
     instructions 1
elif condition 2:
     instructions 2
else:
     instructions 3
```

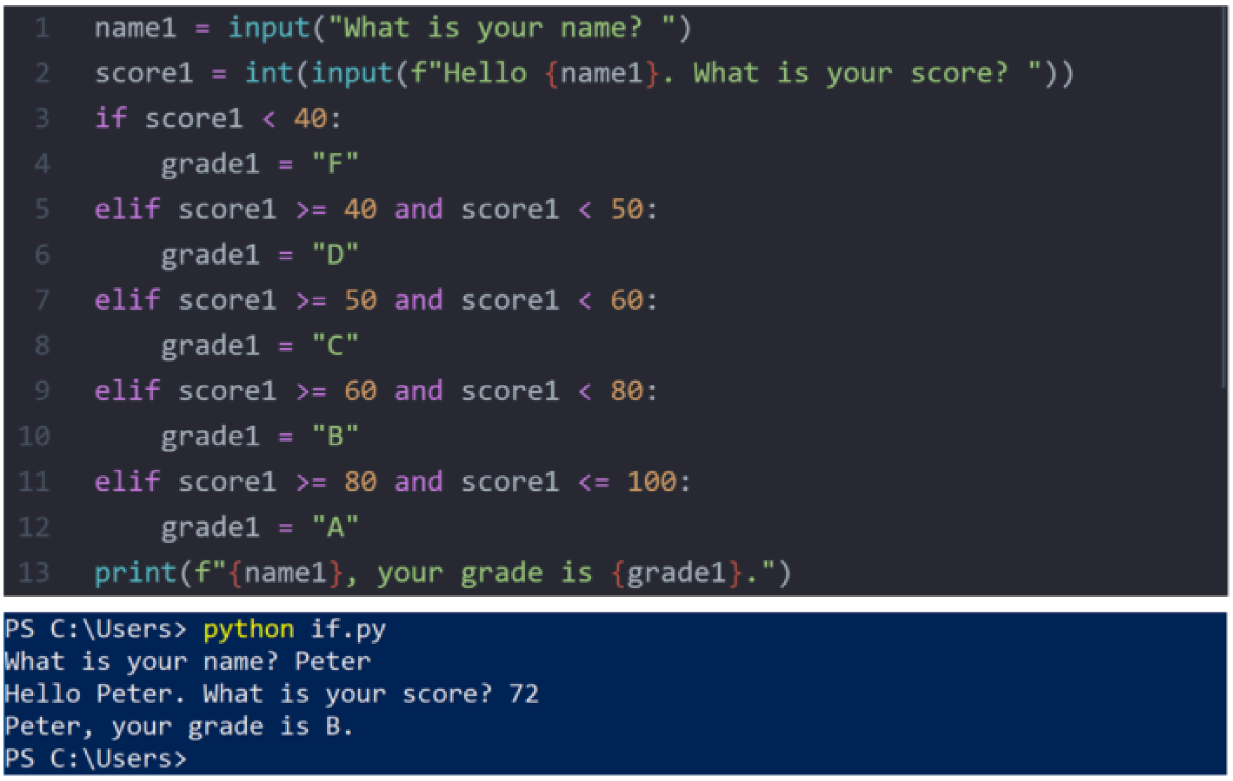
Note that an `if`-`elif`-`else`-block does not necessarily need an `else`-statement. But we should ensure that the conditions being checked by the `if`-statement and the `elif`-statements must cover all possible outcomes, unless we are certain that only those possibilities are being uncovered which do not need any instructions to follow up.

In the example in Figure 1.30, the program is only constructed to separate students into two categories: Pass and fail. It will then print the statement to the user accordingly. Suppose we also give grades to evaluate the performance of the students, we can categorise the scores using `if`-conditions.

If we construct an `if`-block to categorise a numeric variable, we should ensure that the Boolean expressions do not overlap. For instance, if grade A is assigned when the score is between 80 and 100, then 80 should not be included in the condition for getting grade B. The logical operator `and` should be used to indicate the interval for each category since both conditions, namely that the value of the numeric variable must be larger than the lower bound, as well as smaller than the upper bound of the interval, must be fulfilled simultaneously.

In the example shown in Figure 1.32, the `else`-statement has also been omitted since all possible outcomes of the variable `score` have been covered by the `if`-block. Nevertheless, we can also use the `else`-statement instead of the whole elif-condition for grade A if we are confident to do so. Just be cautious that in this case, if certain possibilities were not covered, no instructions would be carried out from the `if`-block, and the behaviour of the subsequent part of the program may be affected.

Example (Cont'd): We will print the grade to the student according to his/her exam score. If a student scores between 80 and 100, his/her grade will be A; and if his/her score is between 80 and 60, he/she will get a B; a score between 50 and 60 is equivalent to grade C; grade D will be given if a student scores between 40 and 50 and any score below 40 belongs to grade F.



```
name1 = input("What is your name? ")
score1 = int(input(f"Hello {name1}. What is your score? "))
if score1 < 40:
    grade1 = "F"
elif score1 >= 40 and score1 < 50:
    grade1 = "D"
elif score1 >= 50 and score1 < 60:
    grade1 = "C"
elif score1 >= 60 and score1 < 80:
    grade1 = "B"
elif score1 >= 80 and score1 <= 100:
    grade1 = "A"
print(f"{name1}, your grade is {grade1}.")
```

```
PS C:\Users> python if.py
What is your name? Peter
Hello Peter. What is your score? 72
Peter, your grade is B.
PS C:\Users>
```

Figure 1.32 Example of `if-elif-else`-Statement

Lastly, the `print()` function is not indented here. As a result, Python interprets it as the part of the code that should be executed after the entire `if`-block and not as part of the instructions following the last condition (`elif score1 >= 80 and score1 <= 100:`).

Read

Read the following three sections of the textbook on conditional statements for control flow:

Exercise 29 What If

Exercise 30 Else and If

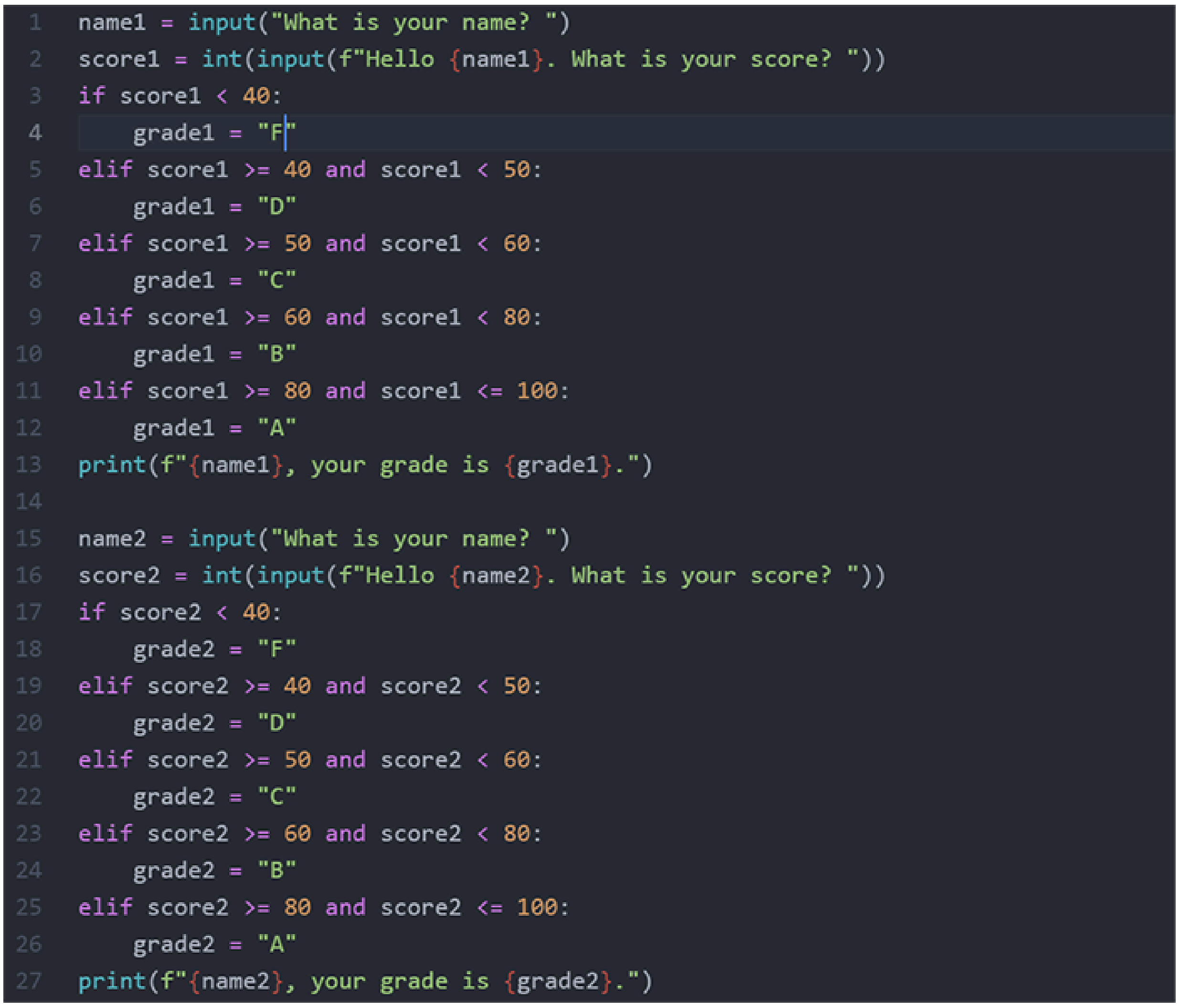
Chapter 5: Loops

Lesson Recording

Loops in Python

5.1 While-Loops

In the student score example, we construct a program in which the name and the mark of one student can be entered. In the early stage of our example, we had scores of two students. If we had to enter their names and then assign a grade to each of them, we would have to repeat the codes in Chapters 2, 3 and 4 twice.



```python
name1 = input("What is your name? ")
score1 = int(input(f"Hello {name1}. What is your score? "))
if score1 < 40:
    grade1 = "F"
elif score1 >= 40 and score1 < 50:
    grade1 = "D"
elif score1 >= 50 and score1 < 60:
    grade1 = "C"
elif score1 >= 60 and score1 < 80:
    grade1 = "B"
elif score1 >= 80 and score1 <= 100:
    grade1 = "A"
print(f"{name1}, your grade is {grade1}.")

name2 = input("What is your name? ")
score2 = int(input(f"Hello {name2}. What is your score? "))
if score2 < 40:
    grade2 = "F"
elif score2 >= 40 and score2 < 50:
    grade2 = "D"
elif score2 >= 50 and score2 < 60:
    grade2 = "C"
elif score2 >= 60 and score2 < 80:
    grade2 = "B"
elif score2 >= 80 and score2 <= 100:
    grade2 = "A"
print(f"{name2}, your grade is {grade2}.")
```

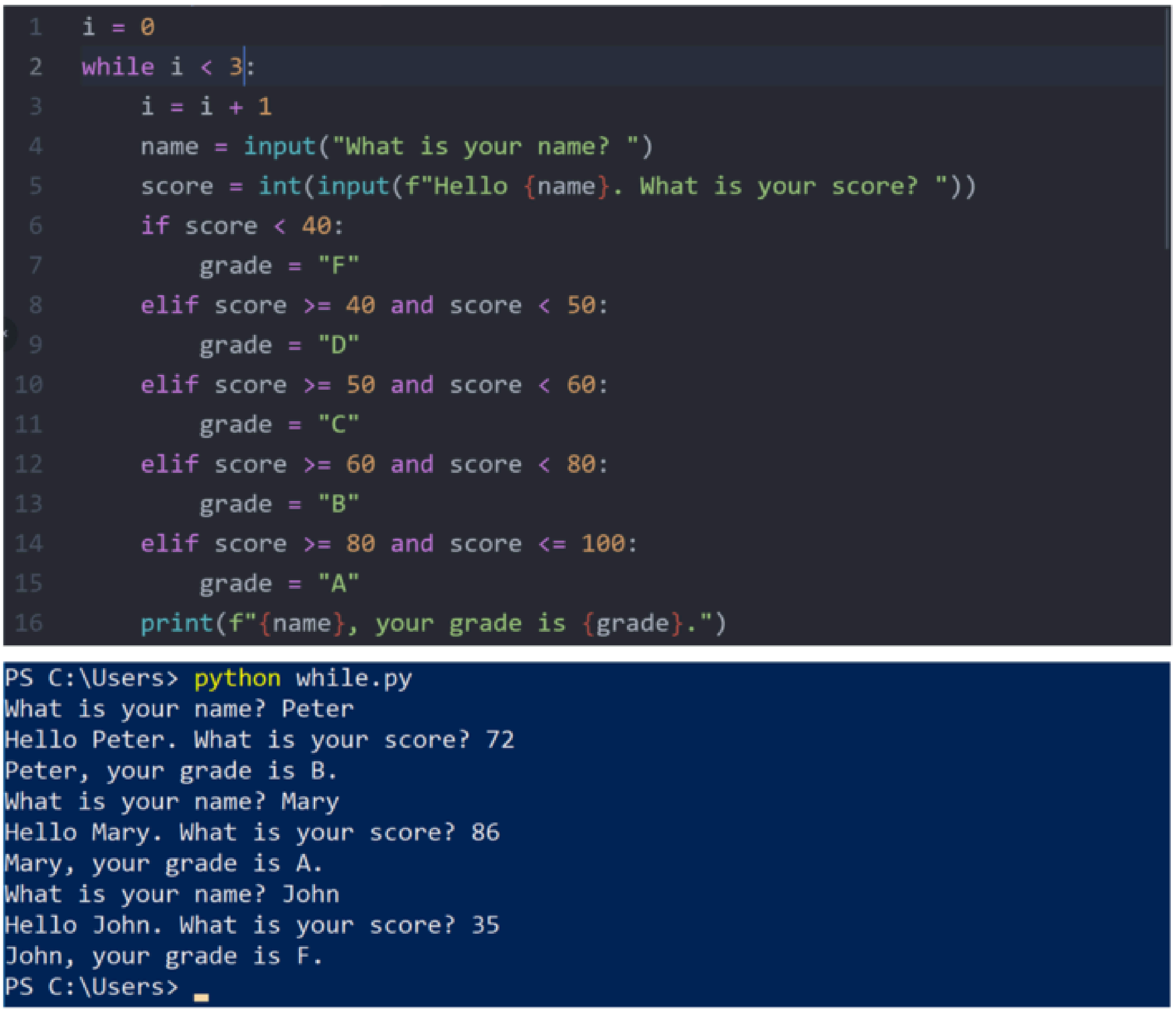
Figure 1.33 Repeating the Same Code for Two Individuals to Enter

Usually, we have more than two students in one class. In order not to expand our codes endlessly and make it clumsy and unreadable, we can construct a loop in our program to repeat the instructions that will be applied for many times. The first type of loops that we will introduce here is the `while`-loop.

```
while conditions:
    instructions
```

The condition is a Boolean expression which controls whether the loop will continue to run for a new iteration or not. If the condition is `True`, Python will go on to execute the instructions that are written with indentation and after the colon behind the conditions. The number of loops can be infinite and will be repeated as long as the condition is `True`. As a result, it is extremely important to ensure that a `while`-loop will be terminated at some stage by fulfilling the exit condition.

Example (Cont'd): We will repeat the previous task for 3 students.



```python
i = 0
while i < 3:
    i = i + 1
    name = input("What is your name? ")
    score = int(input(f"Hello {name}. What is your score? "))
    if score < 40:
        grade = "F"
    elif score >= 40 and score < 50:
        grade = "D"
    elif score >= 50 and score < 60:
        grade = "C"
    elif score >= 60 and score < 80:
        grade = "B"
    elif score >= 80 and score <= 100:
        grade = "A"
    print(f"{name}, your grade is {grade}.")
```

```
PS C:\Users> python while.py
What is your name? Peter
Hello Peter. What is your score? 72
Peter, your grade is B.
What is your name? Mary
Hello Mary. What is your score? 86
Mary, your grade is A.
What is your name? John
Hello John. What is your score? 35
John, your grade is F.
PS C:\Users> _
```

Figure 1.34 Entering Names and Scores Repeatedly by `while`-Loops

Before the loop starts, we initiate a counter variable called `i` with the value 0. This counter will increase by 1 in each iteration. The `while`-loop is set to continue as long as `i` has not reached 3 yet. At the beginning of the last iteration, `i` should be 2 and will become 3 when Python executes the instructions within the `while`-loop. At the end of this iteration, Python will go back to the first line of the `while`-loop, and since the condition `i < 3` is no longer `True`, the program will exit the loop as designed. Therefore, the most crucial command in the whole loop is `i = i + 1`. Without this line, `i` can never reach 3, which is the exit condition, after three iterations. Instead, it will stay at 0, the initial value defined before the loop has started, forever.

Read

Read the following section of the textbook on while loops:

Exercise 33 While Loops

5.2 For-Loops

Another type of loops is the `for`-loop, which are constructed differently. While we need a `True`-condition for a `while`-loop to continue to iterate, we need a list for the running of the `for`-loops. We will discuss the construction and properties of a list in detail in Study Unit 2. Here, we will first learn to generate a simple list of consecutive integers by the `range()` function.

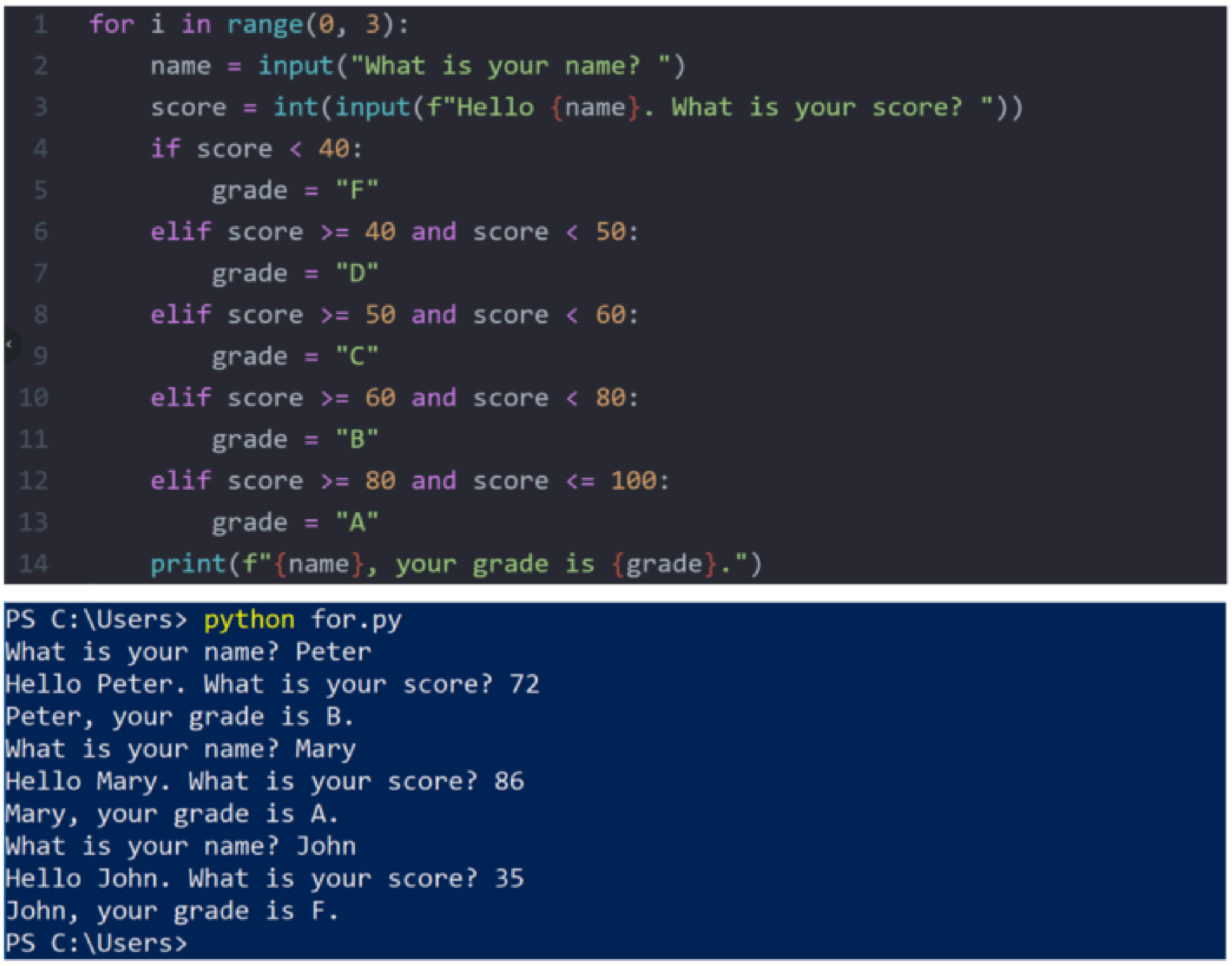
```
variable = range(start, end)
```

The start value can be any integer as long as it is smaller than the end value. Note that the end value is not included in the list. In other words, the list will end at `end - 1`. The generated list of numbers will be assigned to the variable defined on the left-hand side of the syntax. For the `for`-loops, we do not need to store the generated list in a variable first. Instead, we can use the `range()` function in the `for`-statement directly.

```
for counter in range(start, end):
     instructions
```

The `for`-command must end with a colon, followed by the instructions that should be carried out in each iteration. These instructions must be written with indentation. The counter variable will do the counting for the iterations, starting from the `start` value in the first iteration and running through the entire integer list. Once the counter reaches `end – 1`, Python will execute the instructions for the last time and then exit the loop.

Example (Cont'd): We will carry out the previous task using the `for`-loop.



```python
for i in range(0, 3):
    name = input("What is your name? ")
    score = int(input(f"Hello {name}. What is your score? "))
    if score < 40:
        grade = "F"
    elif score >= 40 and score < 50:
        grade = "D"
    elif score >= 50 and score < 60:
        grade = "C"
    elif score >= 60 and score < 80:
        grade = "B"
    elif score >= 80 and score <= 100:
        grade = "A"
    print(f"{name}, your grade is {grade}.")
```

```
PS C:\Users> python for.py
What is your name? Peter
Hello Peter. What is your score? 72
Peter, your grade is B.
What is your name? Mary
Hello Mary. What is your score? 86
Mary, your grade is A.
What is your name? John
Hello John. What is your score? 35
John, your grade is F.
PS C:\Users>
```

Figure 1.35 Entering Names and Scores Repeatedly by `for`-Loops

The `range()` function generates a list containing the values 0, 1, and 2, since 3 will not be included in the list by definition. The code in Figure 1.35 also shows that the counter variable `i` is already integrated in the `for`-statement, and an explicit instruction to increase it by one in each iteration is not required at all.

Read

Read the following section of the textbook on `while`-loops:

Exercise 32 Loops and Lists

5.3 Breaking from Loops

Though we usually have a clear exit condition or a finite list to guarantee a loop to end at a certain point of the program, we may still be confronted with situations where we would like to interrupt the loop and continue with the subsequent program.

In our previous example, suppose we would like to quit the entire program after entering the first student's data due to some reasons, although the program allows us to enter the data for up to three students. It would be reasonable to have a syntax that allows us to break from the loop in a "clean" manner instead of shutting down the computer entirely. The command for such situation is `break`.

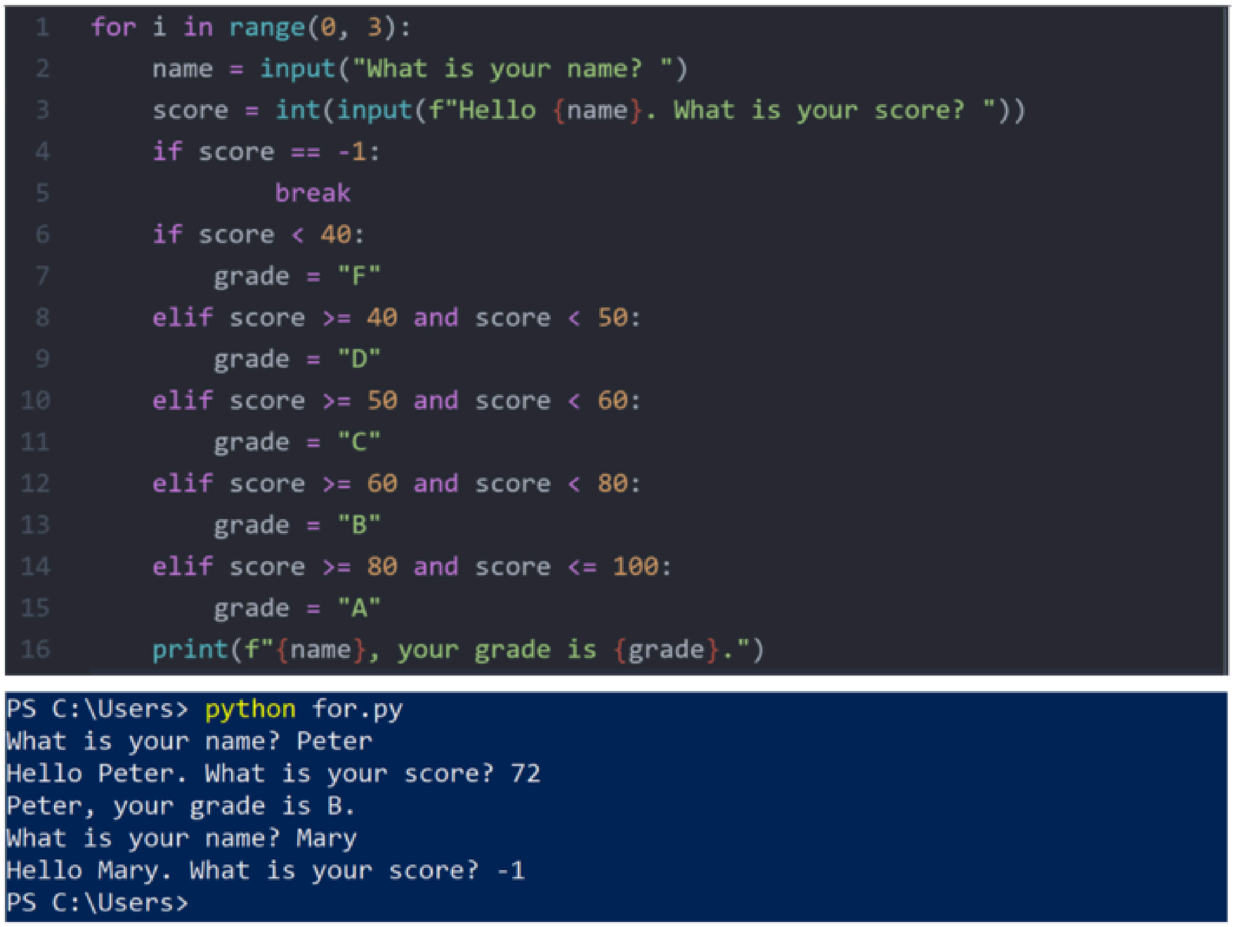
```
for counter in range(start, end):
      …
      if conditions:
           break
```

Equivalently, the `break` command also works within a `while`-loop.

```
while conditions:
     …
     if conditions:
          break
```

Usually, `break` is used together with an `if`-condition since we would only want to break from a loop under some circumstances, and not in general.

Example (Cont'd): Suppose we let the user to quit the program by entering -1 for his/her score now. All we need to do is to add an `if`-condition after the `input()` statement where he/she can enter the score.



```python
for i in range(0, 3):
    name = input("What is your name? ")
    score = int(input(f"Hello {name}. What is your score? "))
    if score == -1:
            break
    if score < 40:
        grade = "F"
    elif score >= 40 and score < 50:
        grade = "D"
    elif score >= 50 and score < 60:
        grade = "C"
    elif score >= 60 and score < 80:
        grade = "B"
    elif score >= 80 and score <= 100:
        grade = "A"
    print(f"{name}, your grade is {grade}.")
```

```
PS C:\Users> python for.py
What is your name? Peter
Hello Peter. What is your score? 72
Peter, your grade is B.
What is your name? Mary
Hello Mary. What is your score? -1
PS C:\Users>
```

Figure 1.36 Breaking from Loops Using `break`

The `break` command is built after the user is asked to enter his/her exam score, but before the grade is being assigned. Basically, the program will still work normally if the `break` command is put after the whole `if-elif`-block. By breaking from the loop before a chunk of codes that will have no further influence on the execution of the program can shorten the running time and make it faster.

Note that we can also apply the `break` command on our example in Figure 1.34 where we use the `while`-loops for the same task instead.

5.4 Error Handling in Input

Another common use of loops is to control the user input following an `input()` statement. For instance, it could happen that the user types in a letter instead of a number for the exam score by accident. In this case, we would like the user to redo the input until it is a number. As a result, we can put the `input()` statement within a `while`-loop and only break from it when the input of the user is valid.

Before we start to construct `while`-loops for user inputs, we have to learn how Python handles errors. In Chapter 3, we mention that if we apply the `int()` function to convert a string variable that contains a non-numeric value to an integer, the program will be interrupted due to value error. And the user will have to restart the program in PowerShell. This can be very annoying if the user only makes a small mistake in one of the input fields, but needs to re-type all the inputs because of the program interruption.

To avoid Python from stopping to execute the program by force, we can use the `try:` syntax.

```
try: #The try-block lets us test a block of code for errors.
    instructions
except exception: #The except-block lets us handle the
 error.
    instructions
else:  #The else-block carries out instructions if no error
 occurs (optional).
    instructions
finally:  #The finally-block executes instructions
 regardless of the result of the try- and except blocks
 (optional).
    instructions
```

The `try-except`-block is an important instrument in Python to handle errors. Basically, we can put any syntax in the `try`-block if we think error can occur in those syntaxes. The `except`-block is to tell Python to continue with the program except for the occurrence of an error, or the occurrence of a specific error that we declare under `exception`. If error indeed occurs, Python will carry out the instructions written after the colon behind the except-statement, instead of stopping the program entirely. The `else`-block and `finally`-block are optional and can be used if we want certain instructions to be carried out if no error occurs or for finalising a `try`-block.

In Python, there are many built-in exceptions. Table 1.7 provides some common ones.

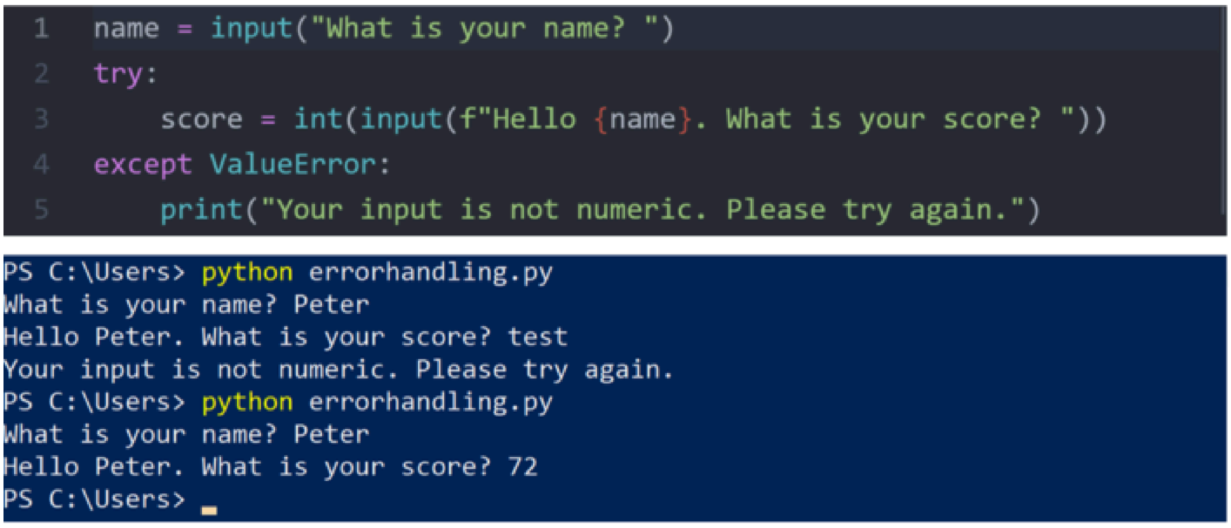
Table 1.7 List of Some Common Exceptions

Exception	Description
`NameError`	Raised when a local or global name is not found. This applies only to unqualified names. The associated value is an error message that includes the name that could not be found.
`TypeError`	Raised when an operation or function is applied to an object of inappropriate type. The associated value is a string giving details about the type mismatch.
`ValueError`	Raised when an operation or function receives an argument that has the right type but an inappropriate value, and the situation is not described by a more precise exception.

(Source: https://docs.python.org/3/library/exceptions.html)

If the user is asked to enter a numeric value such as an exam score, but enter a string instead, we can use `ValueError` as our exception.

Example (Cont'd): We implement a `try`-block to test the validity of score input by the user. If it is a string, the program will warn the user of an erroneous input.



```python
name = input("What is your name? ")
try:
    score = int(input(f"Hello {name}. What is your score? "))
except ValueError:
    print("Your input is not numeric. Please try again.")
```

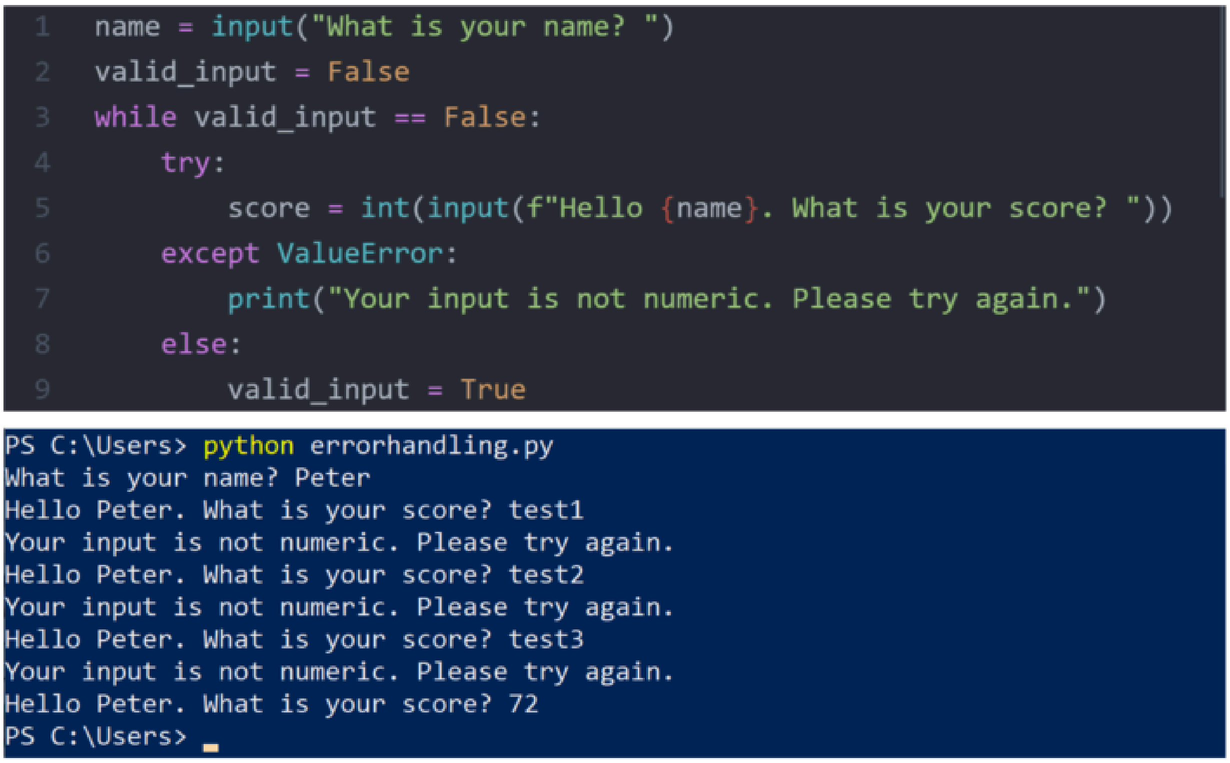
```
PS C:\Users> python errorhandling.py
What is your name? Peter
Hello Peter. What is your score? test
Your input is not numeric. Please try again.
PS C:\Users> python errorhandling.py
What is your name? Peter
Hello Peter. What is your score? 72
PS C:\Users> _
```

Figure 1.37 Using Exception for Error Handling

In the first run, we type in a string "test" where we should actually enter a numeric value for the score. Python detects an error since the syntax tries to use the `int()` function to convert character strings to integers which would usually cause the program to stop running abruptly. From the output, we can see that Python prints the warning message we put in the `print()` function to the screen and ends the program "properly" as if no error has occurred. In the second run, we enter a numeric value as required. Python detects no error and simply skips the `except`-block.

After defining the `try`-block to instruct Python on how to handle errors, we can construct a `while`-loop around it. As condition for the loop to continue iterating is when a Boolean variable that indicates a valid input does not change from `False` to `True`. Hence, if this Boolean variable is `True`, the program will break from the loop.

Example (Cont'd): Now we put the entire `try`-block within a `while`-loop. The `while`-loop will stop iterating once the input for `score` is numeric. If it is non-numeric, the user will see a warning message and he will also be asked to re-enter his exam score. The whole procedure will last until the input is numeric.



```python
name = input("What is your name? ")
valid_input = False
while valid_input == False:
    try:
        score = int(input(f"Hello {name}. What is your score? "))
    except ValueError:
        print("Your input is not numeric. Please try again.")
    else:
        valid_input = True
```

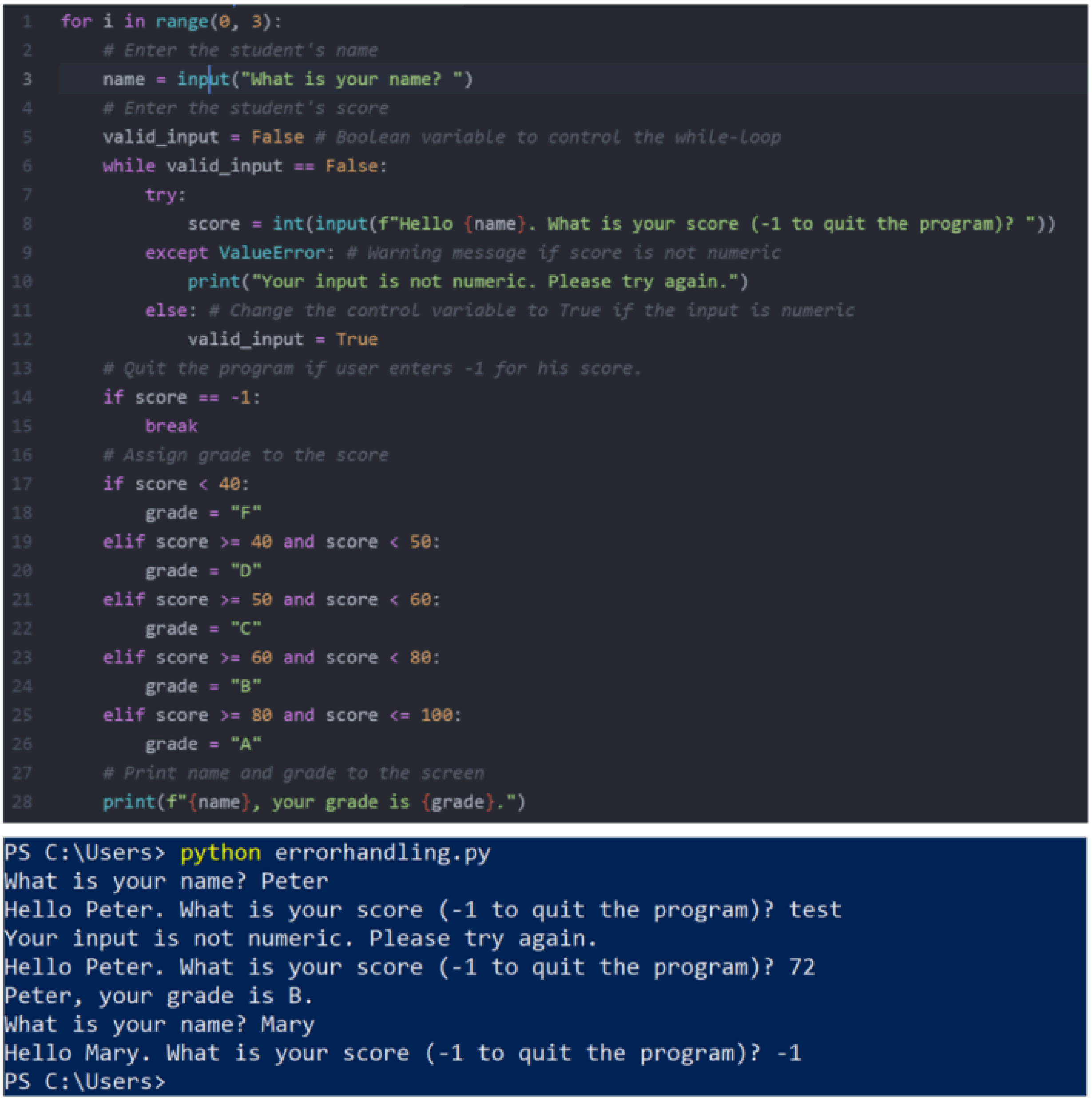
```
PS C:\Users> python errorhandling.py
What is your name? Peter
Hello Peter. What is your score? test1
Your input is not numeric. Please try again.
Hello Peter. What is your score? test2
Your input is not numeric. Please try again.
Hello Peter. What is your score? test3
Your input is not numeric. Please try again.
Hello Peter. What is your score? 72
PS C:\Users>
```

Figure 1.38 Using `while`-loop for User Input If Error Occurs

We initiate a Boolean variable called `valid_input` before the `while`-loop starts. The initial value of this variable is `False` which we also use as the condition for the `while`-loop to continue to iterate. In the `try`-block, we add an `else`-statement for the case that the input is correct, and the follow-up instruction here is to change the value of `valid_input` to `True` so that the `while`-loop stops. Note that we can use the `break` command here as well. In the output screen, we can see that if the input is non-numeric, the program will print a warning message in a new line and then ask the user to re-enter the score until the input is valid.

It is possible or even desirable to set a maximum number of input trials in order not to have the program running endlessly. A counter variable can be added to the loop, and the program will exit the loop after the maximum allowed number of iterations has been reached.

Example (Cont'd): The complete program of this study unit containing all the techniques we have learned is given in the following figure.



```python
for i in range(0, 3):
    # Enter the student's name
    name = input("What is your name? ")
    # Enter the student's score
    valid_input = False # Boolean variable to control the while-loop
    while valid_input == False:
        try:
            score = int(input(f"Hello {name}. What is your score (-1 to quit the program)? "))
        except ValueError: # Warning message if score is not numeric
            print("Your input is not numeric. Please try again.")
        else: # Change the control variable to True if the input is numeric
            valid_input = True
    # Quit the program if user enters -1 for his score.
    if score == -1:
        break
    # Assign grade to the score
    if score < 40:
        grade = "F"
    elif score >= 40 and score < 50:
        grade = "D"
    elif score >= 50 and score < 60:
        grade = "C"
    elif score >= 60 and score < 80:
        grade = "B"
    elif score >= 80 and score <= 100:
        grade = "A"
    # Print name and grade to the screen
    print(f"{name}, your grade is {grade}.")
```

```
PS C:\Users> python errorhandling.py
What is your name? Peter
Hello Peter. What is your score (-1 to quit the program)? test
Your input is not numeric. Please try again.
Hello Peter. What is your score (-1 to quit the program)? 72
Peter, your grade is B.
What is your name? Mary
Hello Mary. What is your score (-1 to quit the program)? -1
PS C:\Users>
```

Figure 1.39 The Complete Example Program of Study Unit 1

Summary

We have learned the basics of writing and executing Python programs. We have also been introduced to the various variable types and some operators that can be applied to them. We have then discussed how to generate screen output and how to let the user enter answers and assign them as values to the variables. Furthermore, we have covered the construction of conditional statements to dynamically change the program behaviour if necessary. Finally, we have also come to know the use of loops to repeat routine tasks for an endless number of times within a program.

Formative Assessment

1. From which directory (folder) can you run your Python script that is saved with the .py extension?

   a. From the folder where Python is installed

   b. From the folder where the operating system is installed

   c. From the folder where I have saved my Python script

   d. From any arbitrary folder

2. Which line of code is not a valid Python syntax?

   a. `print("This is {a} wrong syntax!")`

   b. `0 = 0`

   c. `0 == 1`

   d. `y = a / int(b)`

3. Which of the following is a valid Python variable name?

   a. `iamavariablelol`

   b. `:)iamavariable`

   c. `007imavariable`

   d. `i-am-a-variable-lol`

4. What is the value of `int(-0.5)` in Python?

   a. -1

   b. 0.5

   c. 0

   d. 1

5. What does the `.format()` method do?

   a. It formats the font of the string in a `print()` function.

b. It replaces the expression in a curly bracket within a string by its value.

c. It replaces the curly bracket by a round bracket in a string.

d. It replaces the curly bracket within a string by the value of the expression in the `.format()` method.

6. Which of the following Boolean expression is false?

   a. `-5 <= 0`

   b. `10 ** 2 == 100`

   c. `int(-0.5) < int(-0.2)`

   d. `18 / 3 > 1 + 4`

7. Which Boolean expression is equivalent to `not(a and b)`?

   a. `not a and not b`

   b. `not a or not b`

   c. `not a and b`

   d. `a or b`

8. Which statement is correct regarding the `if-elif-else` statement block?

   a. The `if-elif-else` statement must end with an `end`-statement.

   b. There must be an `else`-statement in every `if-elif-else` statement block.

   c. `If-elif-else` statement block can also start with an `elif`- or `else`-statement.

   d. Behind each `if`-, `elif`-, and `else`-statement must be a colon before the instruction block starts.

9. Which values will be printed on the screen given the following Python code?

```
counter = 0
while counter <= 3:
     print(counter)
     counter = counter + 1
```

  a. 0, 1, 2, 3
  b. 1, 2, 3, 4
  c. 0, 1, 2, 3, 4
  d. 0, 2, 3

10. Which of the following is correct about `for`-loops in Python?
    a. We need to initiate a counter before the loop starts.
    b. We must write a line to increase the counter by one within the loop.
    c. We need an exit condition for the loop.
    d. We can use the `range()` function to generate a list of numbers as the index of the `for`-loop iterations.

Solutions or Suggested Answers

Formative Assessment

1. From which directory (folder) can you run your Python script that is saved with the .py extension?

   a. From the folder where Python is installed

   Incorrect. Python would not be able to find your Python script there unless you have saved it in the Python program folder, which is rather unlikely.

   b. From the folder where the operating system is installed

   Incorrect. Python would not be able to find your Python script there unless you have saved it in the system folder, which is also rather unlikely.

   c. From the folder where I have saved my Python script

   Correct. You must change to the folder where you have saved your Python script so that Python can find your file.

   d. From any arbitrary folder

   Incorrect. Python would not be able to find your Python script unless you are accidentally in the folder where you have saved your Python script.

2. Which line of code is not a valid Python syntax?

   a. `print("This is {a} wrong syntax!")`

   Incorrect. The syntax is correct since the curly bracket and its content will be treated as part of the printing string.

   b. `0 = 0`

Correct. The left-hand side of a value-assignment syntax must be a variable name. A number there is invalid since Python interprets it as assigning a value to a number.

c. `0 == 1`

Incorrect. This syntax is valid since it is a Boolean expression.

d. `y = a / int(b)`

Incorrect. This is a valid syntax since we can carry out a division in which the denominator is being converted to an integer.

3. Which of the following is a valid Python variable name?

a. `iamavariablelol`

Correct. This variable name is valid since it starts with a character and contains no invalid character.

b. `:)iamavariable`

Incorrect. This variable name is invalid since it starts with a colon and contains invalid characters such as closing round bracket.

c. `007imavariable`

Incorrect. This variable name is invalid since it starts with a number.

d. `i-am-a-variable-lol`

Incorrect. This variable name contains invalid characters such as hyphen.

4. What is the value of `int(-0.5)` in Python?

a. -1

Incorrect. The `int()` function does not round down a negative number.

b. 0.5

Incorrect. The `int()` function is not used to convert a value to absolute number.

c. 0

Correct. The `int()` function will take away all the decimal places.

d. 1

Incorrect. The `int()` function does not round down a negative number and then convert it to absolute number.

5. What does the `.format()` method do?

a. It formats the font of the string in a `print()` function.

Incorrect. The `.format()` method does not format the font of a string.

b. It replaces the expression in a curly bracket within a string by its value.

Incorrect. This would be done by the `f`-instruction in the `print()` function for string formatting.

c. It replaces the curly bracket by a round bracket in a string.

Incorrect. The `.format()` method does not replace the curly bracket by a round bracket in a string unless the code explicitly requires Python to do so.

d. It replaces the curly bracket within a string by the value of the expression in the `.format()` method.

Correct. The value of the expression in the `.format()` method will be used to replace the curly bracket within the printing string.

6. Which of the following Boolean expression is false?

a. `-5 <= 0`

Incorrect. The expression is true since it requires -5 to be smaller or equal to 0.

b. `10 ** 2 == 100`

Incorrect. Since `10 ** 2 = 100`, the left-hand side and the right-hand side are equal.

c. `int(-0.5) < int(-0.2)`

Correct. Since `int(-0.5) is 0 and int(-0.2)` is also 0, `0 < 0` is a false relation.

d. `18 / 3 > 1 + 4`

Incorrect. Since 18 / 3 = 6 and 1 + 4 = 5, `6 > 5` is a true relation.

7. Which Boolean expression is equivalent to `not(a and b)`?

a. `not a and not b`

Incorrect. `not a` and `not b` is equivalent to `not(a or b)`

b. `not a or not b`

Correct. `not a` or `not b` is equivalent to `not(a and b)`

c. `not a and b`

Incorrect. `not a and b` is equivalent to `not(a or not b)`

d. `a or b`

Incorrect. `a` or `b` is equivalent to `not(not a and not b)`

8. Which statement is correct regarding the `if-elif-else` statement block?

a. The `if-elif-else` statement must end with an `end`-statement.

Incorrect. No `end`-statement is needed for an `if-elif-else` statement block.

b. There must be an `else`-statement in every `if-elif-else` statement block.

Incorrect. An `if-elif-else` statement block does not necessarily require an `else`-statement.

c. `If-elif-else` statement block can also start with an `elif`- or `else`-statement.

Incorrect. An `if-elif-else` statement block must start with an `if`-statement.

d. Behind each `if`-, `elif`-, and `else`-statement must be a colon before the instruction block starts.

Correct. Behind every if-, elif-, and else-statement must be a colon.

9. Which values will be printed on the screen given the following Python code?

```
counter = 0
while counter <= 3:
     print(counter)
     counter = counter + 1
```

a. 0, 1, 2, 3

Correct. Since the `print()` function comes before the increment of the counter, the counter will be printed starting from its initial value 0 and goes until 3 with an increase of 1 in each iteration.

b. 1, 2, 3, 4

Incorrect. Since the `print()` function comes before the increment of the counter, the counter will be printed starting from its initial value 0 and not 1.

c. 0, 1, 2, 3, 4

Incorrect. Since the loop will only continue to run if the value in counter is smaller or equal to 3, 4 cannot be printed based on this code.

d. 0, 2, 3

Incorrect. Since the increment of the counter can only be 1 for each iteration of the loop, a jump from 0 to 2 is impossible based on this code.

10. Which of the following is correct about `for`-loops in Python?

    a. We need to initiate a counter before the loop starts.

    Incorrect. We only need to initiate a counter before a `while`-loop starts.

    b. We must write a line to increase the counter by one within the loop.

    Incorrect. We only need to write a line to increase the counter by one within a `while`-loop.

    c. We need an exit condition for the loop.

    Incorrect. A `for`-loop does not need any exit condition since it runs through a finite list. Once the list comes to an end, Python will exit the loop.

    d. We can use the `range()` function to generate a list of numbers as the index of the `for`-loop iterations.

    Correct. Every `for`-loop needs a list for it to run through. One type of list is a list of integers that can serve as the index for the for-loop iterations, and we can generate such a list by the range() function.

References

Atom.io. (n.d.). *A hackable text editor for the 21st Century*. GitHub. https://atom.io

Learn Python. (n.d.). *Python – Basic operators*. tutorialspoint. https://www.tutorialspoint.com/python/python_basic_operators.htm

Python Tutorial. (n.d.). *Python 3 escape sequences*. Python-ds.com. https://www.python-ds.com/python-3-escape-sequences

Python Tutorial. (n.d.). *Python variables*. w3schools.com. https://www.w3schools.com/python/python_variables.asp

Python.org. (n.d.). *Built-in exceptions*. Python Software Foundation. https://docs.python.org/3/library/exceptions.html

Python.org. (n.d.). *Built-in types*. Python Software Foundation. https://docs.python.org/3/library/stdtypes.html#numeric-types-int-float-complex

Python.org. (n.d.). *Download python*. Python Software Foundation. https://www.python.org/downloads/

Shaw, Z. A. (2017). *Learn python 3 the hard way*. Addison-Wesley Professional.

Study Unit 2

Data Types and Functions

Learning Outcomes

By the end of this unit, you should be able to:

1. Differentiate the various types of compound data structures in Python
2. Discuss how Python manages packages, modules, functions, etc.

Overview

Python provides numerous compound data types to group a collection of values together. Compound data structures organise and store data in a way that they can be accessed and worked with efficiently. These structures also define the relationship between the data and the operations that can be performed on them. We will learn in this study unit how to create, when to use, and what operations to perform on the most common Python compound data structures: tuples, lists and dictionaries. Furthermore, we will also learn about what functions and methods are and how they can be integrated in our program. Lastly, we will also deal with Python packages and modules which are, together with functions and methods, very useful for reusing and extending our tools of performing specific tasks in Python programming.

Chapter 1: Tuples, Lists, Dictionaries

1.1 Tuples

Lesson Recording

Python Tuples

1.1.1 Defining Tuples

In Study Unit 1, we have learned that we could store values in a variable for later operations in our code. The type of a variable depends on whether we want to store numeric values or character strings in it. More often, we work with multiple data points of the same nature such as the names of all students in a class. It would be inconvenient to create a new python variable for each data point like, e.g., `student1`, `student2`, etc. What we can do instead is to store all data points in some kind of compound data structure.

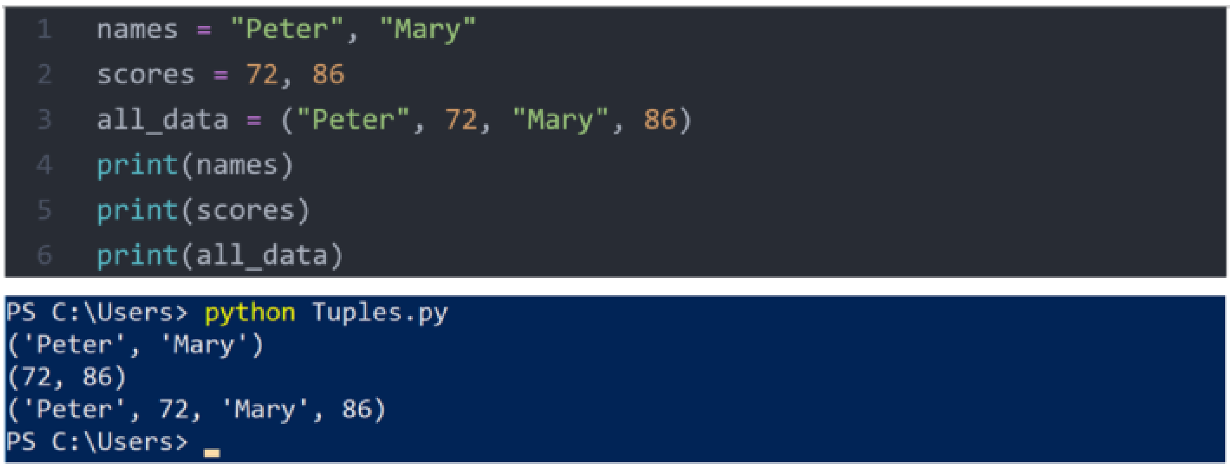
Python facilitates several types of compound data for our use. One of these data types is the tuple. A tuple is a collection of values written as comma-separated items between a pair of round brackets, similar like vectors in mathematics. Unlike vectors, tuples are not specifically designed for mathematical operations. The items, or elements in a tuple can be numeric, string, or a mixture of both.

```
tuple_name = (element1, element2, …)
```

Note that tuples are immutable, that is, we are not allowed to modify them once they are defined. Nevertheless, they are more efficient in terms of performance and memory use.

Tuples are useful in situations where we want to share the data with other users without granting them the permission to edit the content.

Example (Students' score, cont'd): We assign the names of two students to a tuple called `names`, the `scores` of the same students to a tuple scores and a mixture of the names and the scores to `all_data`.



```python
names = "Peter", "Mary"
scores = 72, 86
all_data = ("Peter", 72, "Mary", 86)
print(names)
print(scores)
print(all_data)
```

```
PS C:\Users> python Tuples.py
('Peter', 'Mary')
(72, 86)
('Peter', 72, 'Mary', 86)
PS C:\Users> _
```

Figure 2.1 Definition of Tuples

We can assign elements to a tuple with or without writing them in parentheses in our code, as long as they are separated by commas. Nevertheless, when printing them on the screen, they will be wrapped by a pair of round brackets. Furthermore, the tuple `all_data`, which consists of two strings and two integers, shows us that tuples can be indeed a mixture of strings, integers, and floats.

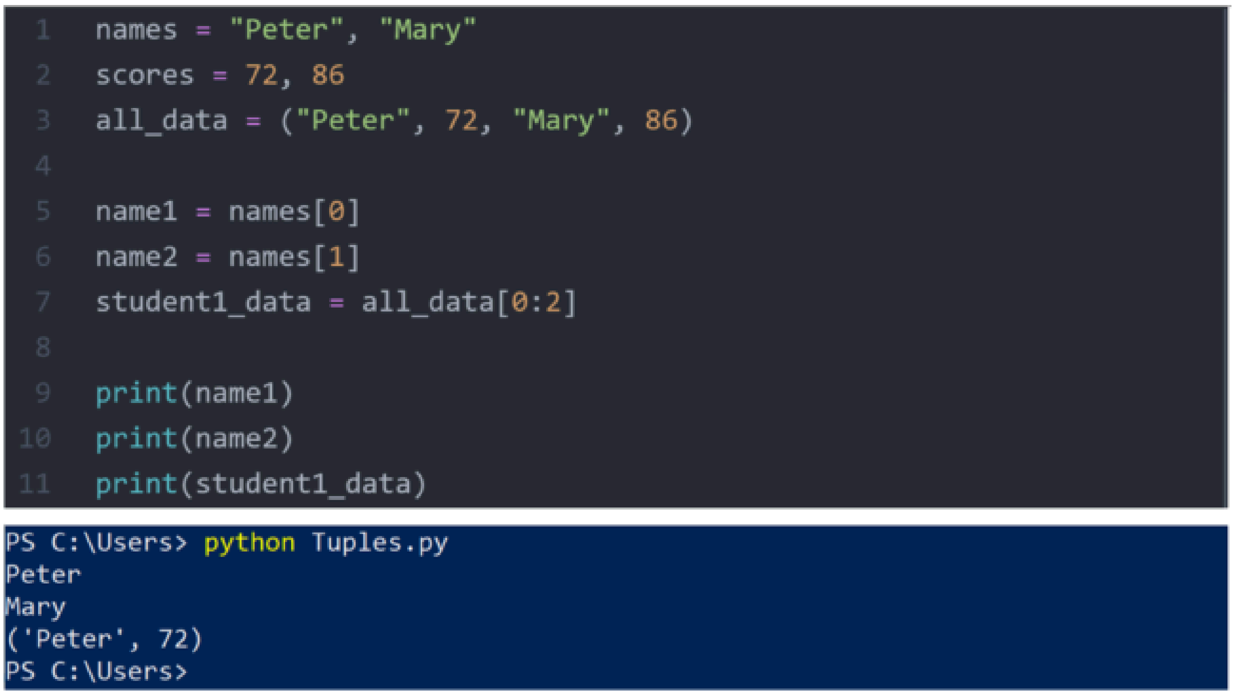
1.1.2 Subsetting Tuples and Indexing

To access one specific element in a tuple, we need to use the index operator `[]`. In Python, the index of any compound data type, i.e., tuples, lists and dictionaries, begins with 0 and ends with the total number of elements minus 1. In other words, the index for the first element in a tuple is 0, the index for the second element is 1, and the third one is 2, and so on. If we want to subset more than a single element from a tuple, we can put the start and end indices in the index operator, connecting them with a colon.

```
tuple_subset = tuple_name[start:end]
```

It is important to recall that the index `end` will *not* be included in the subsetting procedure.

Example (Cont'd): We would now like to extract the first element from the tuple `names` and store it in a variable called `name1`, the second element in `name2`, as well as the entire data of the first student including his name and score from the tuple `all_data` in a new tuple called `student1_data`.



```
names = "Peter", "Mary"
scores = 72, 86
all_data = ("Peter", 72, "Mary", 86)

name1 = names[0]
name2 = names[1]
student1_data = all_data[0:2]

print(name1)
print(name2)
print(student1_data)
```

```
PS C:\Users> python Tuples.py
Peter
Mary
('Peter', 72)
PS C:\Users>
```

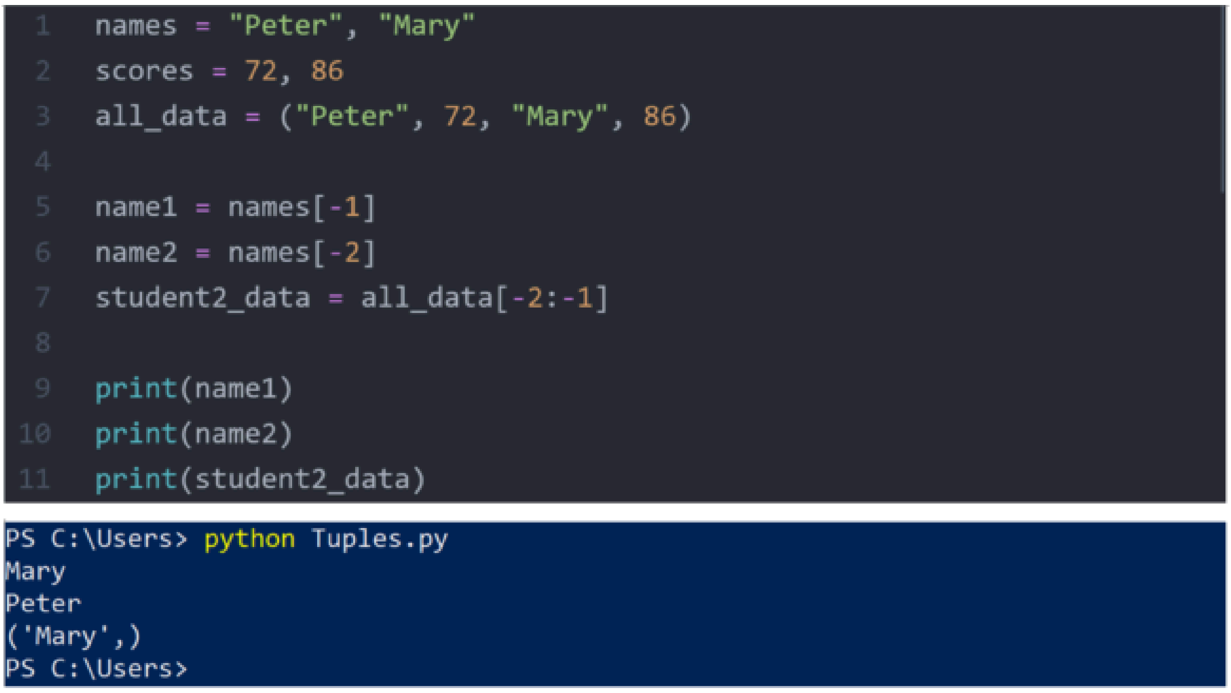
Figure 2.2 Subsetting Tuples

If we subset a single element from a tuple, it will become a variable of the type that corresponds to the type of the extracted data. In this case, the variables `name1` and `name2` are strings since "Peter" and "Mary" are stored as string in the original tuple names. And if we subset multiple elements from a tuple, the result will be a tuple as well and the data type of each element will also be taken over from the original tuple.

In the above example, the tuple `student1_data` should be a subset of the tuple `all_data` with the element indexed `0:2`, where the index 2 is excluded. In other word, it only contains the elements 0 and 1, which are the first and second elements: Peter and 72.

We can also use negative indices to access the elements of a tuple starting from the last element. That is, the index -1 indicates the last element, -2 the second last, etc.

Example (Cont'd): We would now like to extract the last element from the tuple `names` and store it in a variable called `name1`, the second last element in `name2`, as well as the entire data of the second student including his name and score from the tuple `all_data` in a new tuple called `student2_data`.

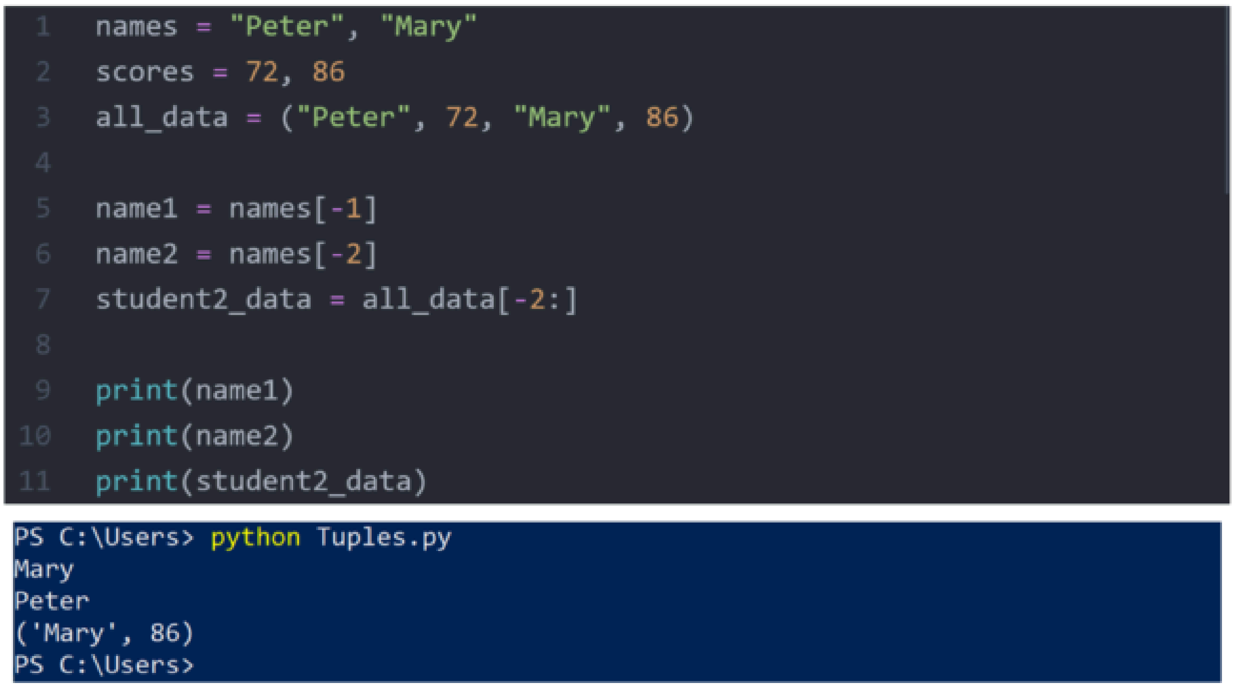


```
names = "Peter", "Mary"
scores = 72, 86
all_data = ("Peter", 72, "Mary", 86)

name1 = names[-1]
name2 = names[-2]
student2_data = all_data[-2:-1]

print(name1)
print(name2)
print(student2_data)
```

```
PS C:\Users> python Tuples.py
Mary
Peter
('Mary',)
PS C:\Users>
```

Figure 2.3 Subsetting of Tuples with Negative Indices

Everything seems alright in the above output except that the last element of the tuple `student2_data`, the value 86, is missing. The reason is that the last index is never included. In our example, the subsetting indices we wish to have are -2 and -1. But since Python does not include -1, we only receive `student2_data[-2]`, which is "Mary" in this case. To overcome this dilemma, we need to leave the end index blank after the colon. Python will interpret it as "take all indices until the end".



```
names = "Peter", "Mary"
scores = 72, 86
all_data = ("Peter", 72, "Mary", 86)

name1 = names[-1]
name2 = names[-2]
student2_data = all_data[-2:]

print(name1)
print(name2)
print(student2_data)
```

```
PS C:\Users> python Tuples.py
Mary
Peter
('Mary', 86)
PS C:\Users>
```

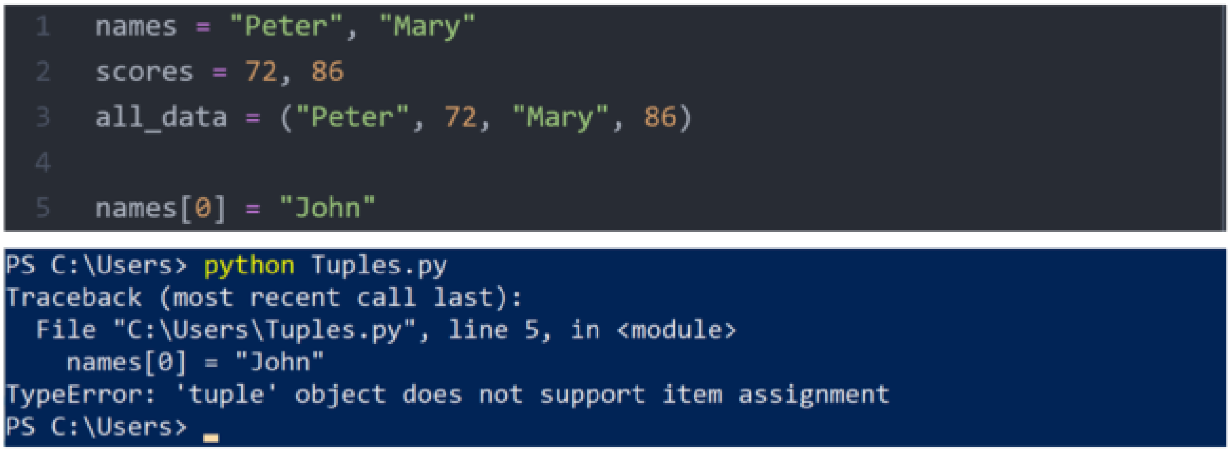
Figure 2.4 Subsetting Tuples with "Open End" Indexing

This is the correct output that we want to obtain originally. Note that the "open end" indexing also works for positive indices.

1.1.3 Concatenating Tuples

Though we can access the tuple elements by using the index operator, we are neither allowed to change the values of it nor to add a new value to an existing tuple.

Example (Cont'd): Suppose we would like to change the first element in the tuple `names` from "Peter" to "John", we will receive an error message as result.

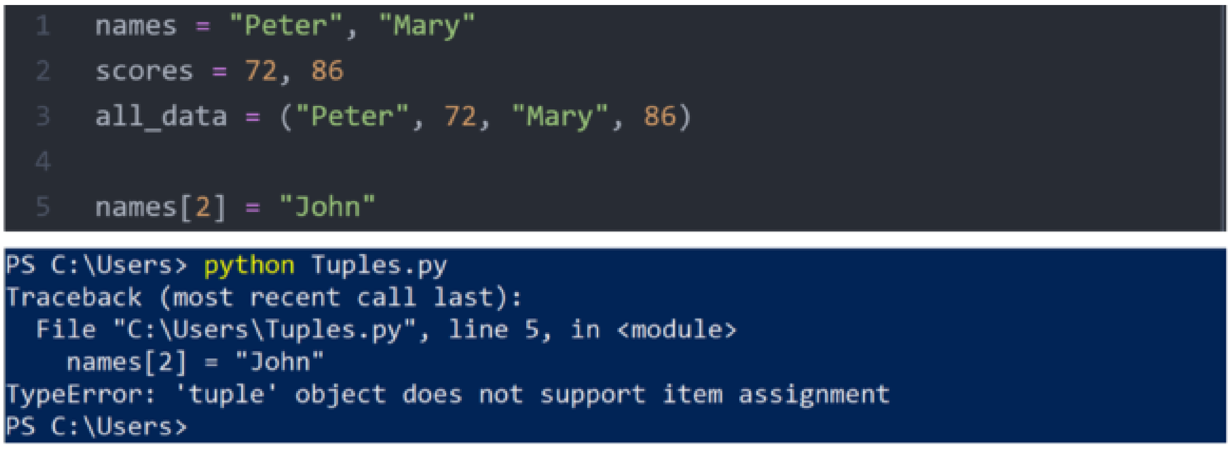


```
names = "Peter", "Mary"
scores = 72, 86
all_data = ("Peter", 72, "Mary", 86)

names[0] = "John"
```

```
PS C:\Users> python Tuples.py
Traceback (most recent call last):
  File "C:\Users\Tuples.py", line 5, in <module>
    names[0] = "John"
TypeError: 'tuple' object does not support item assignment
PS C:\Users>
```

Figure 2.5 Erroneous Modification of Tuples

We will now try to add the name "John" to the tuple `names`. Since `names` has 2 elements, the index of the last element must be 1. Hence, we add the new element to `names` by referring to a new index, namely 2.



```
names = "Peter", "Mary"
scores = 72, 86
all_data = ("Peter", 72, "Mary", 86)

names[2] = "John"
```

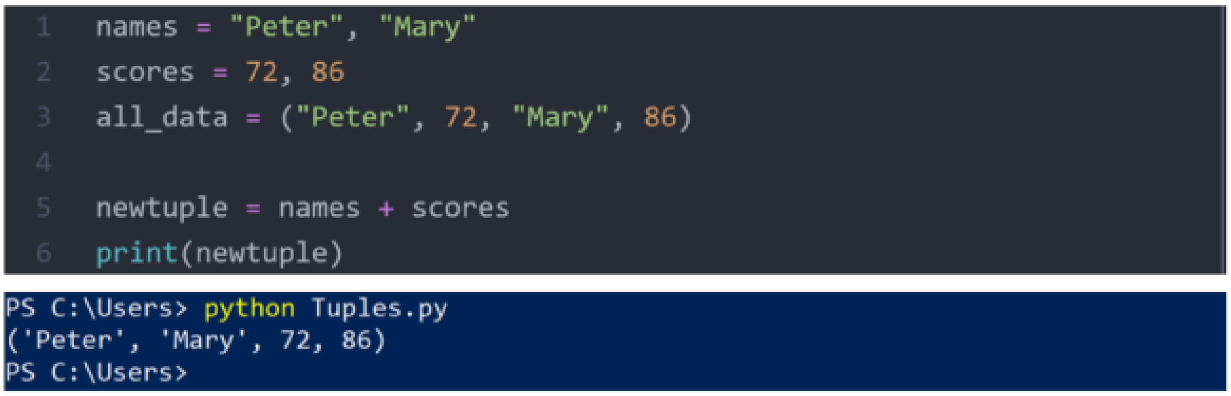
```
PS C:\Users> python Tuples.py
Traceback (most recent call last):
  File "C:\Users\Tuples.py", line 5, in <module>
    names[2] = "John"
TypeError: 'tuple' object does not support item assignment
PS C:\Users>
```

Figure 2.6 Erroneous Adding of Elements to Tuples

Unsurprisingly, we also receive an error message here.

Nevertheless, we are allowed to concatenate two tuples into a single tuple by connecting them with a "+" sign.

Example (Cont'd): Suppose we would like to concatenate the two tuples `names` and `scores` and name the new one `newtuple`.

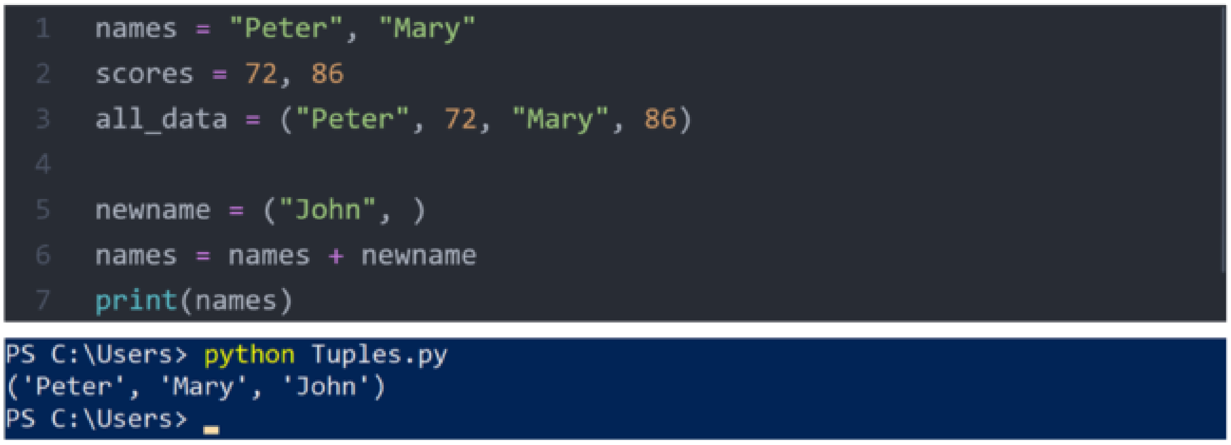


```
names = "Peter", "Mary"
scores = 72, 86
all_data = ("Peter", 72, "Mary", 86)

newtuple = names + scores
print(newtuple)
```

```
PS C:\Users> python Tuples.py
('Peter', 'Mary', 72, 86)
PS C:\Users>
```

Figure 2.7 Concatenation of Two Tuples

The concatenation of tuples works in the same way as it works for strings. As a result, we can re-attempt to add a new name `"John"` to the tuple `names`. The only thing we need to change is to put `"John"` in a tuple first.



```
names = "Peter", "Mary"
scores = 72, 86
all_data = ("Peter", 72, "Mary", 86)

newname = ("John", )
names = names + newname
print(names)
```

```
PS C:\Users> python Tuples.py
('Peter', 'Mary', 'John')
PS C:\Users> _
```

Figure 2.8 Adding Elements to Tuples by Concatenation

To put the new name "John" into a tuple is indeed tricky since Python would not recognise syntaxes such as `("John")` or `"John"` as a tuple. The reason here is that all elements in a tuple must be separated by commas. As a result, we put a comma behind `"John"` and leave the next element blank so as to tell Python that this is a tuple of length 1. We can see that after concatenating `newname` with names, Python will ignore the blank element in `newname` and append `"John"` as the only element in `newname` to `"Peter"` and `"Mary"`, the original elements in `names`.

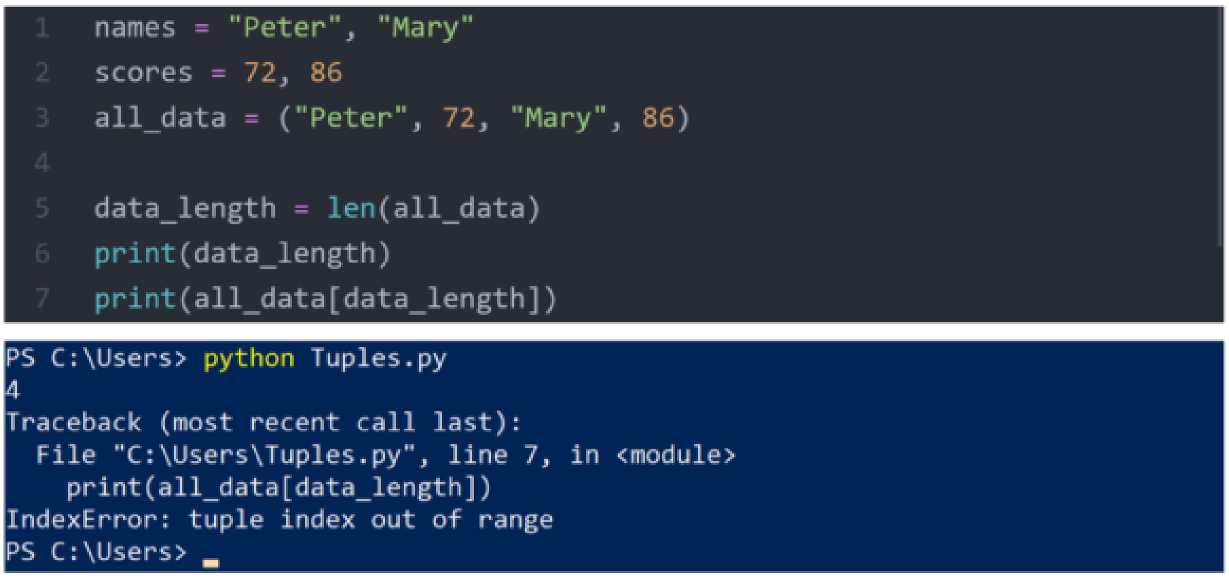
1.1.4 Length of Tuples

When dealing with tuples, lists or dictionaries, the function `len()` can be useful since it will return the length, i.e., the number of elements of such an object.

```
tuple_length = len(tuple_name)
```

The length is also often used as an index to subset a tuple or control the indexing in our code so that we cannot refer to indices that go beyond the largest index of a tuple.

Example (Cont'd): We use the `len()` function to determine the length of the tuple `all_data`. We will then use it as an index to subset the last element of a tuple.

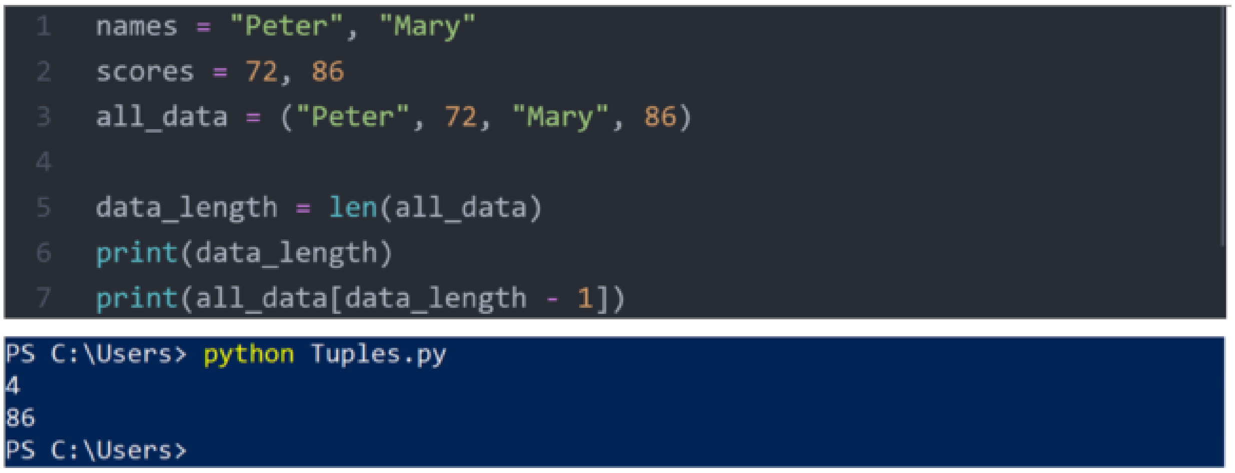


```python
names = "Peter", "Mary"
scores = 72, 86
all_data = ("Peter", 72, "Mary", 86)

data_length = len(all_data)
print(data_length)
print(all_data[data_length])
```

```
PS C:\Users> python Tuples.py
4
Traceback (most recent call last):
  File "C:\Users\Tuples.py", line 7, in <module>
    print(all_data[data_length])
IndexError: tuple index out of range
PS C:\Users> _
```

Figure 2.9 Length of Tuples and Index Out of Range

We receive an error message here since `all_data[data_length]` is equal to `all_data[4]`. Recall that the last index of a tuple is the length of the tuple minus 1. As a result, we can reach our original goal by subtracting 1 from the `data_length`, which is equal to 4, for indexing.



```python
names = "Peter", "Mary"
scores = 72, 86
all_data = ("Peter", 72, "Mary", 86)

data_length = len(all_data)
print(data_length)
print(all_data[data_length - 1])
```

```
PS C:\Users> python Tuples.py
4
86
PS C:\Users>
```

Figure 2.10 Subsetting the Last Element of Tuples

By using the index `data_length - 1`, we are now accessing the element `all_data[3]`, which is the last element of the tuple `all_data` and 86 in this case.

1.1.5 For-Loops and Tuples

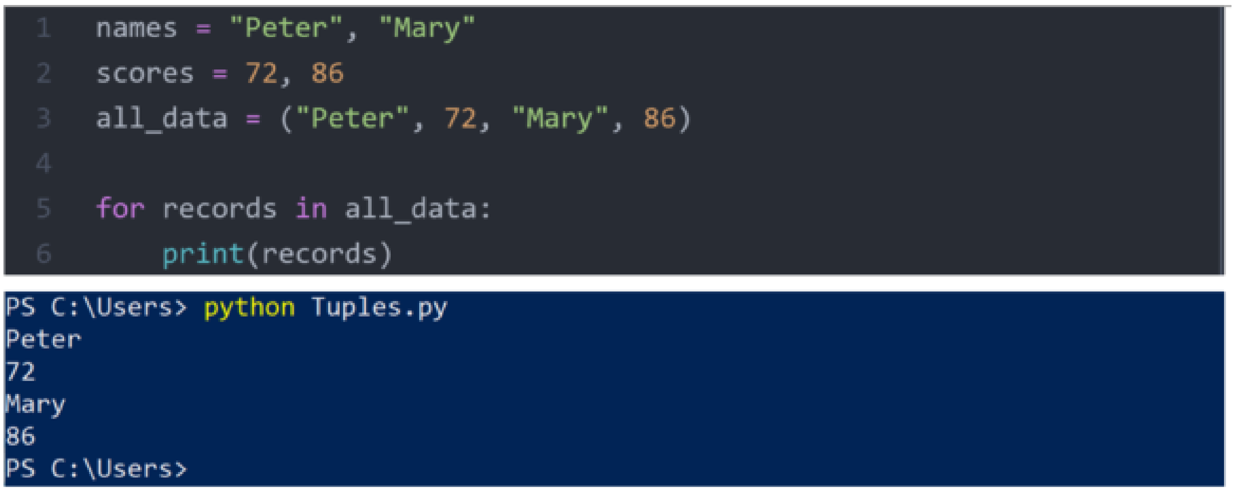
One advantage of tuples, or other compound data types, is that we can access and extract their elements by using the `for`-loop iteratively.

In Study Unit 1, we learned how to use the `range()` function to generate a list for the `for`-loop to run over it. Generally, if there are tuples, lists or dictionaries already created and existing while the program is running, we can use the `for`-loop directly by putting the name of the tuple, list, or dictionary in the `for`-statement.

```
for counter in tuple_name:
     instructions
```

In the above syntax, `tuple_name` can certainly be replaced by any `list_name` or `dictionary_name`.

Example (Cont'd): Now we would like to print out each element of the tuple `all_data` onto the screen. The name of the counter here is `records`.



```python
names = "Peter", "Mary"
scores = 72, 86
all_data = ("Peter", 72, "Mary", 86)

for records in all_data:
    print(records)
```

```
PS C:\Users> python Tuples.py
Peter
72
Mary
86
PS C:\Users>
```

Figure 2.11 Printing Tuple Elements by `for`-Loops

1.2 Lists

Lesson Recording

Python Lists

1.2.1 Creating Lists

Another type of compound data type in Python is the list. A list is just like a tuple but with two main differences:

i. The data are comma-separated items wrapped by a pair of square brackets.

ii. The content of a list is modifiable.

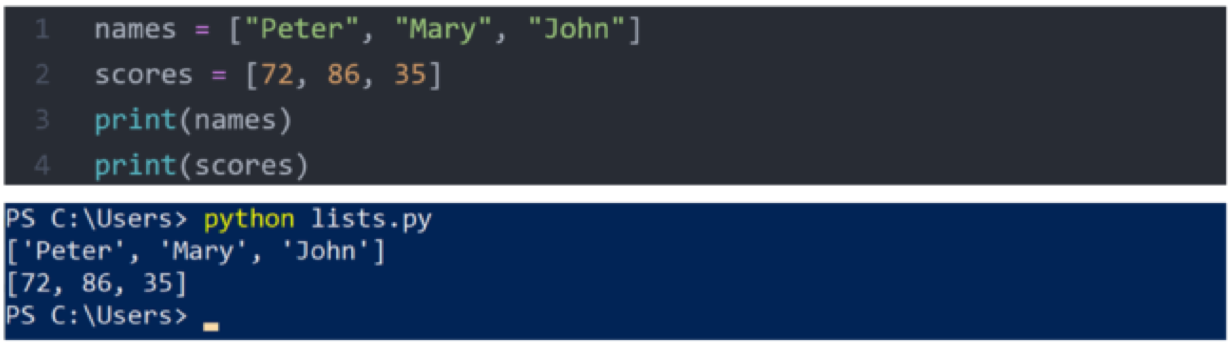
We can construct a Python list in a similar fashion like a tuple.

```
list_name = [element1, element2, …]
```

As mentioned before, we must wrap the data in a list by square brackets. However, unlike a tuple which we may omit the round brackets when defining it in our code as long as the data are separated by commas, we must define a list with the square brackets in the program. If we omit the brackets, Python will interpret the data as a tuple.

Same as tuples, lists may contain any type of values: floats, integer, Booleans, strings, or more advanced Python types like lists. The last one is indeed a very interesting property of the Python compound data type since we can namely put a list within a list, or a tuple in a dictionary, and so on.

Example (Cont'd): We define two lists, `names` and `scores`, to store the data of the students in the class.



```
names = ["Peter", "Mary", "John"]
scores = [72, 86, 35]
print(names)
print(scores)
```

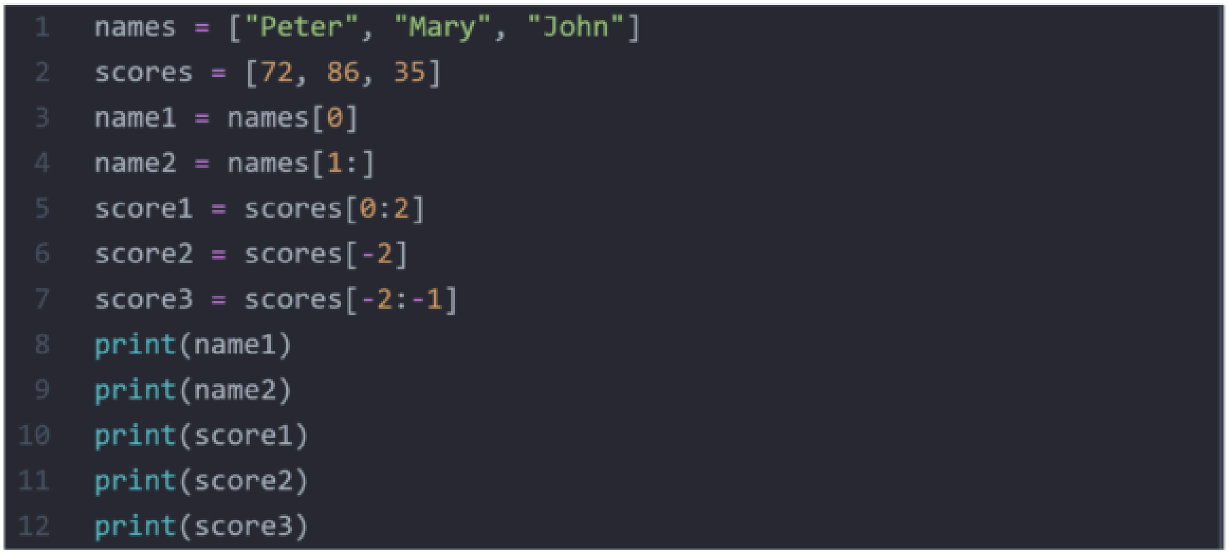
```
PS C:\Users> python lists.py
['Peter', 'Mary', 'John']
[72, 86, 35]
PS C:\Users>
```

Figure 2.12 Creating Lists in Python

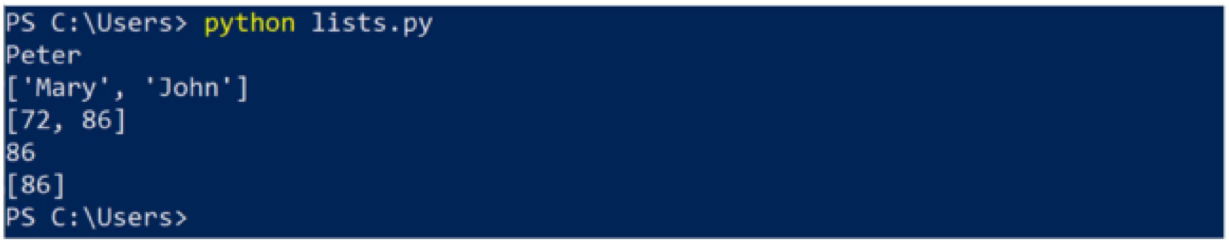
1.2.2 Subsetting Lists

We can also access elements of a list by the index operator. All the indexing and subsetting techniques introduced in Chapter 1.1.2 for tuples are applicable to lists.

Example (Cont’d): Here we will subset the lists that we have defined above into four new lists and variables, respectively.



```
 1   names = ["Peter", "Mary", "John"]
 2   scores = [72, 86, 35]
 3   name1 = names[0]
 4   name2 = names[1:]
 5   score1 = scores[0:2]
 6   score2 = scores[-2]
 7   score3 = scores[-2:-1]
 8   print(name1)
 9   print(name2)
10   print(score1)
11   print(score2)
12   print(score3)
```



```
PS C:\Users> python lists.py
Peter
['Mary', 'John']
[72, 86]
86
[86]
PS C:\Users>
```

Figure 2.13 Subsetting Lists

In the first line (line 1 and 2 are not counted here), `name1 = names[0]`, we simply extract the first element of `names` and store it in `name1`. Since we have only extracted one element, it will not be stored as a new list. Instead, it will become a variable, which we can see from the screen output that the value `"Peter"` is not wrapped by a pair of square brackets. And the type of the new variable will be the same as the type of the extracted element, which is a string variable in this case.

In the second line, `name2 = names[1:]`, we extract multiple elements from the original list and save it as a new list called `name2`. Recall that if we use open end indexing, that is, we leave the value behind the colon blank, Python will take all elements from the original list starting from the starting value until the end. Here, it indicates that Python should extract the elements with the indices 1 and 2 from `names`.

In the third line, `score1 = scores[0:2]`, the first and second elements from `scores` will be extracted and saved into a new list `score1`. We should always be aware that the last index is never included in indexing. As a result, only the elements 72 and 86, and not 35 are taken over in `score1`.

In the fourth line, `score2 = scores[-2]`, we use negative indexing to extract element starting from the last element of `scores`. The value -2 indicates that we would like to extract the second last element, which is 86 in this case. The resulting object here is indeed a numeric variable since we only extract one numeric value from the original list.

In the fifth line, `score3 = scores[-2:-1]`, we extract the element with the indices `-2:-1` from `scores`. Since the last index is not included in the subsetting, the only relevant index here is -2. As a result, we should obtain the same result as the fourth line. However, there is one significant difference between this line and the fourth line: the new object here is still a list instead of a variable. In other words, if we want to create a new list with a single element from another list, we will have to use multiple indexing.

1.2.3 Modifying Lists

Different from tuples, we can change the content of a list or add elements to it. To edit specific items of a list, we simply assign new values to them after subsetting them.

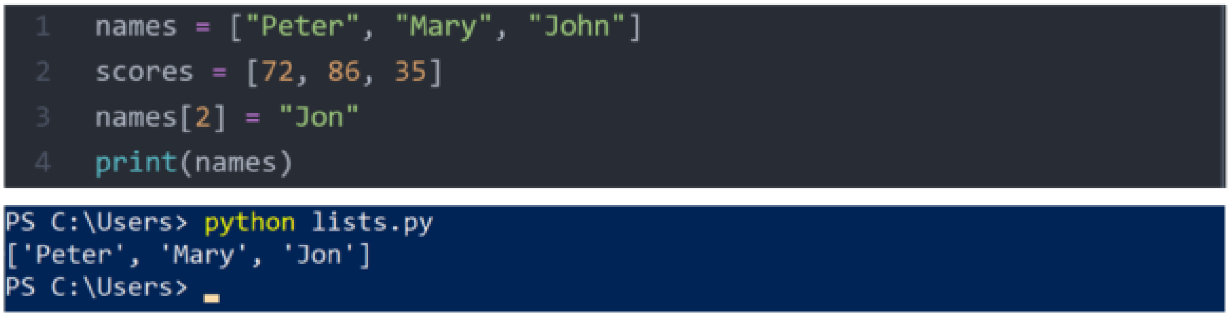
```
list_name[index] = new value
```

We can also use multiple indexing to edit multiple items.

```
list_name[start index:end index] = [list with new values]
```

If we only intend to edit the items in the list, then the length of the indices on the left-hand side, which is the number of items to be modified, must be the same as the length of the list with the new values. If the list of new values is longer than the indices, items will be added to the list on the left-hand side. On the other hand, if the list of indices is shorter, items will be removed from the original list.

Example (Cont'd): Suppose the name of the third student is not John, but Jon, and we would like to change it now. Once again, the third item has the index 2.

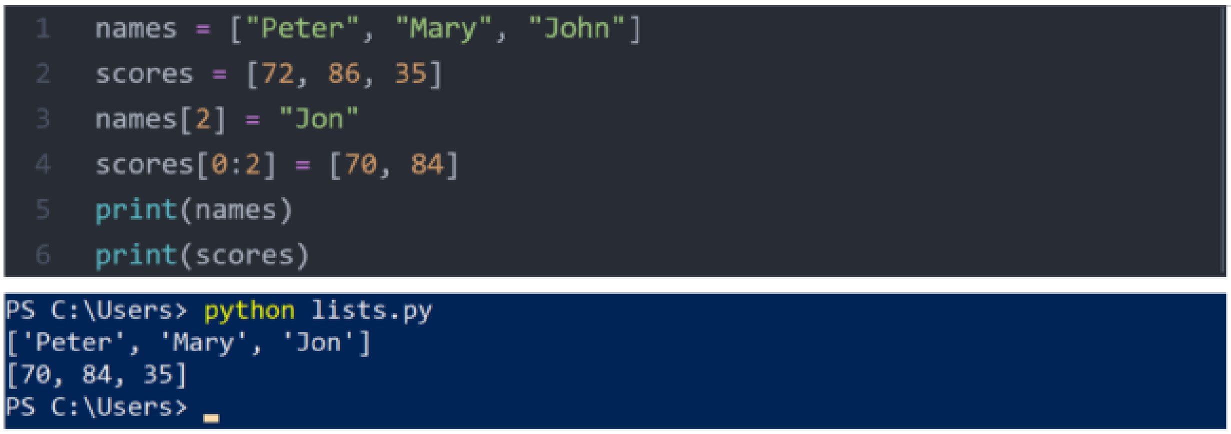


```
names = ["Peter", "Mary", "John"]
scores = [72, 86, 35]
names[2] = "Jon"
print(names)
```

```
PS C:\Users> python lists.py
['Peter', 'Mary', 'Jon']
PS C:\Users> _
```

Figure 2.14 Editing Single Item in a List

Figure 2.14 shows how the third element has been replaced by a new value. Now, suppose that the lecturer has to deduct two marks from the students who scored 72 and 86 after rechecking their exam papers. In other words, we need to modify the first two elements of the list `scores`.

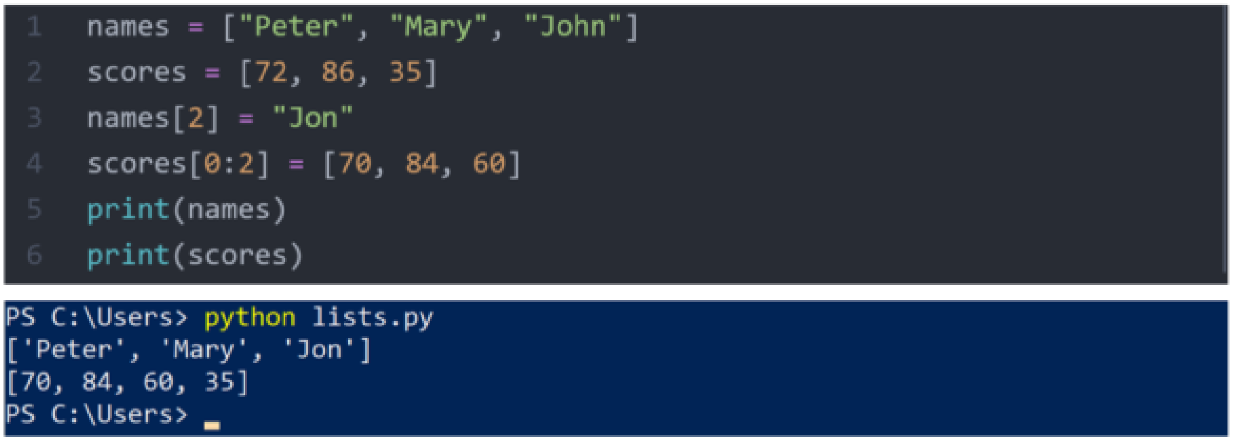


```
names = ["Peter", "Mary", "John"]
scores = [72, 86, 35]
names[2] = "Jon"
scores[0:2] = [70, 84]
print(names)
print(scores)
```

```
PS C:\Users> python lists.py
['Peter', 'Mary', 'Jon']
[70, 84, 35]
PS C:\Users> _
```

Figure 2.15 Editing Multiple Items in a List

In the second line of the output, the first value has been replaced by 70 and the second one by 84. However, if our list `[70, 84]`, does not only contain two values but three, the output will become:



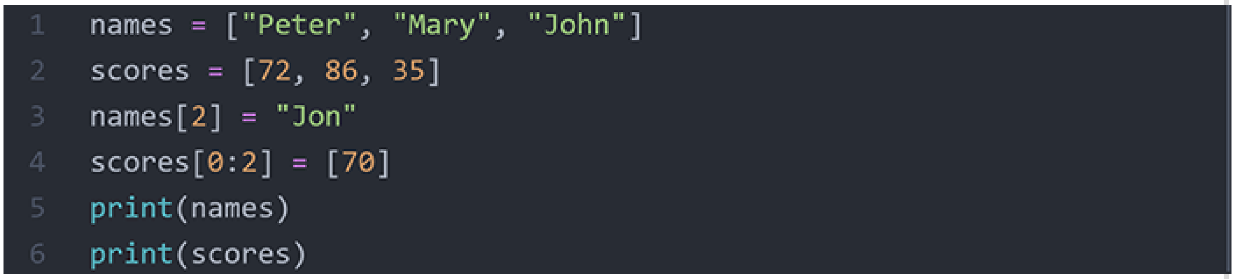
```
names = ["Peter", "Mary", "John"]
scores = [72, 86, 35]
names[2] = "Jon"
scores[0:2] = [70, 84, 60]
print(names)
print(scores)
```

```
PS C:\Users> python lists.py
['Peter', 'Mary', 'Jon']
[70, 84, 60, 35]
PS C:\Users> _
```

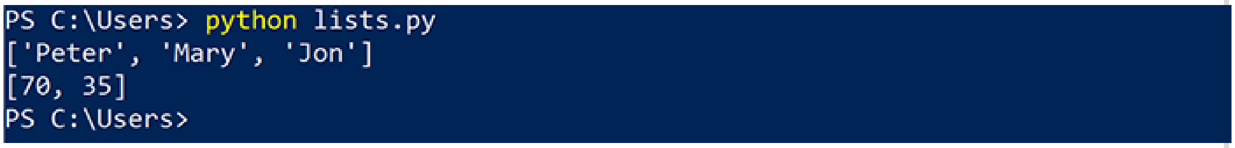
Figure 2.16 Replacing More Values than the Specified Indices

Python will not treat it as a syntax error. Instead, it will replace the values of the indices 0 and 1 by the first two new values, 70 and 84, and add a new value to it, namely 60, before returning to the rest of the original list. As a result, the list `scores` has now four instead of three elements.

Let us turn to the opposite situation and assume that the indices for modification are longer than the list of new values.



```python
names = ["Peter", "Mary", "John"]
scores = [72, 86, 35]
names[2] = "Jon"
scores[0:2] = [70]
print(names)
print(scores)
```



```
PS C:\Users> python lists.py
['Peter', 'Mary', 'Jon']
[70, 35]
PS C:\Users>
```

Same as before, Python will not treat the code here as a syntax error as well. Instead, it will replace the values of the indices 0 and 1 by the only new value, 70, and then return to the rest of the original list, which is 35, the last value. As a result, the list `scores` has now two instead of three elements.

In the above examples, we can see that lists can be extended or shrunk with the replacement of certain items in the original list by a longer or shorter list of new values.

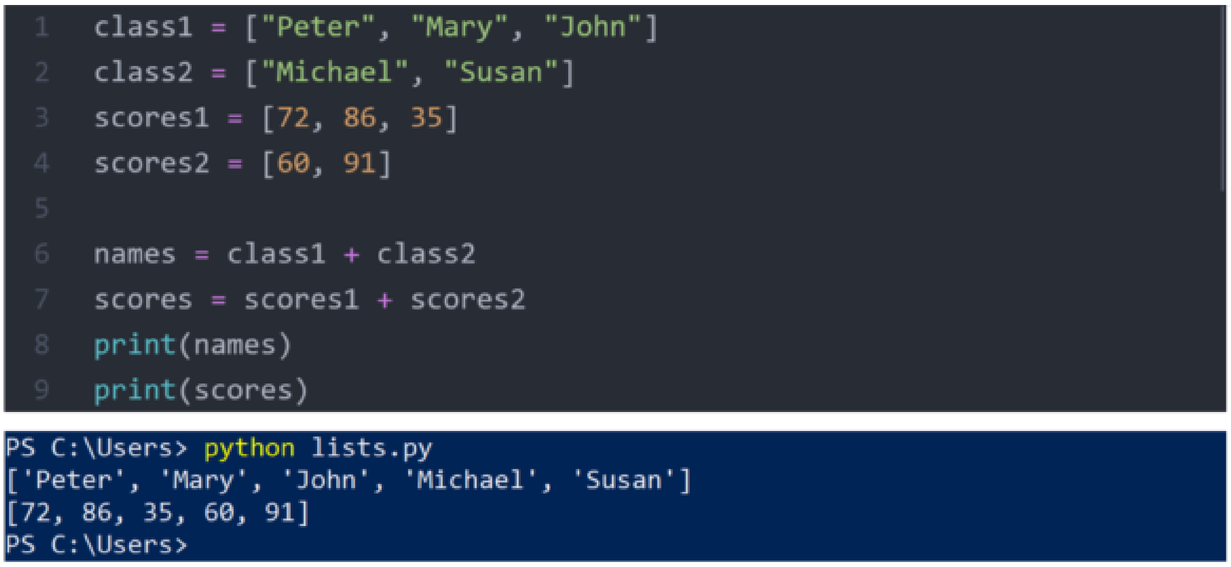
1.2.4 Concatenating & Merging Lists

Same as tuples, we can concatenate multiple lists into one by "adding" them together.

```
combined_list = list1 + list2 + …
```

Basically, Python uses the addition operation "+" to concatenate objects such as strings, tuples, or lists, rather than to carry out arithmetic addition except for numeric values. Concatenating lists is a recommended step in Python programming if the nature, i.e., content and type, of the lists is identical.

Example (Cont'd): Suppose we have two classes for the same course. The student names of these classes are stored in the lists `class1` and `class2`, respectively. Equivalently, the exam scores are stored in the lists scores1 and scores2. Now, we would like to concatenate `class1` and `class2` into a new list called `names` and `scores1` and `scores2` into a new list called `scores`.



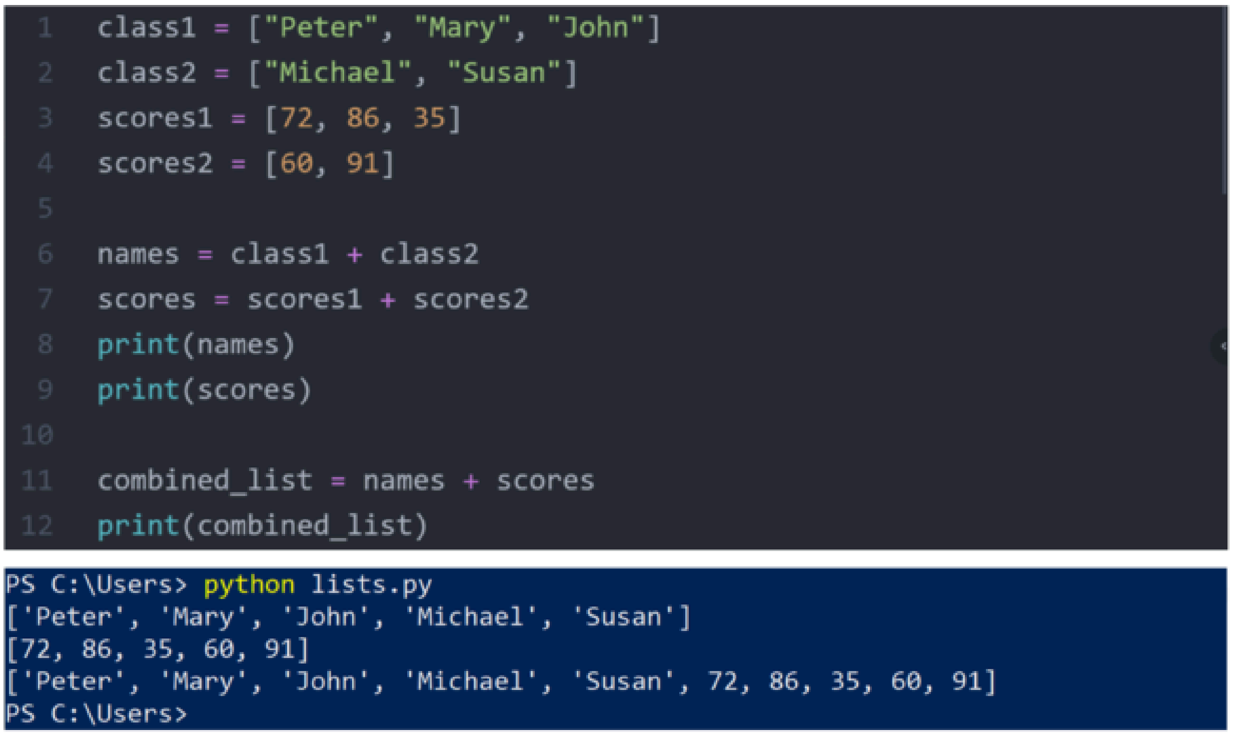
```python
class1 = ["Peter", "Mary", "John"]
class2 = ["Michael", "Susan"]
scores1 = [72, 86, 35]
scores2 = [60, 91]

names = class1 + class2
scores = scores1 + scores2
print(names)
print(scores)
```

```
PS C:\Users> python lists.py
['Peter', 'Mary', 'John', 'Michael', 'Susan']
[72, 86, 35, 60, 91]
PS C:\Users>
```

Figure 2.17 Concatenating Lists

Even if the nature of the two lists is not identical, it is sometimes still quite convenient to store their data in one list for later use.

Example (Cont'd): Suppose we would like to concatenate the two lists `names` and `scores` into a new list called `combined_list` in order to store the data of the entire class in a single compound data source.



```
class1 = ["Peter", "Mary", "John"]
class2 = ["Michael", "Susan"]
scores1 = [72, 86, 35]
scores2 = [60, 91]

names = class1 + class2
scores = scores1 + scores2
print(names)
print(scores)

combined_list = names + scores
print(combined_list)
```

```
PS C:\Users> python lists.py
['Peter', 'Mary', 'John', 'Michael', 'Susan']
[72, 86, 35, 60, 91]
['Peter', 'Mary', 'John', 'Michael', 'Susan', 72, 86, 35, 60, 91]
PS C:\Users>
```

Figure 2.18 Concatenating Lists with Different Data Types

So, the new list consists of the ten elements combined from the two lists `names` and `scores`. Apparently, concatenating lists with different types of data, such as strings and integers in this example, is allowed in Python.

Figure 2.18 shows a straightforward concatenation of two lists in Python. However, such a combination of the two lists will lead to difficulty to distinguish the original nature such as meanings and types of the elements. For instance, in the above example, suppose the first item in the list `scores` is the exam score of the first student in `names`, that is, Peter's exam score is 72, we will not be able to assign the score to the corresponding name unless we know that each score always belongs to the name five positions before. If we keep in mind that the length and contents of the concatenated list may change every now and then, the meaning and source of each element will become more and more untraceable with time.

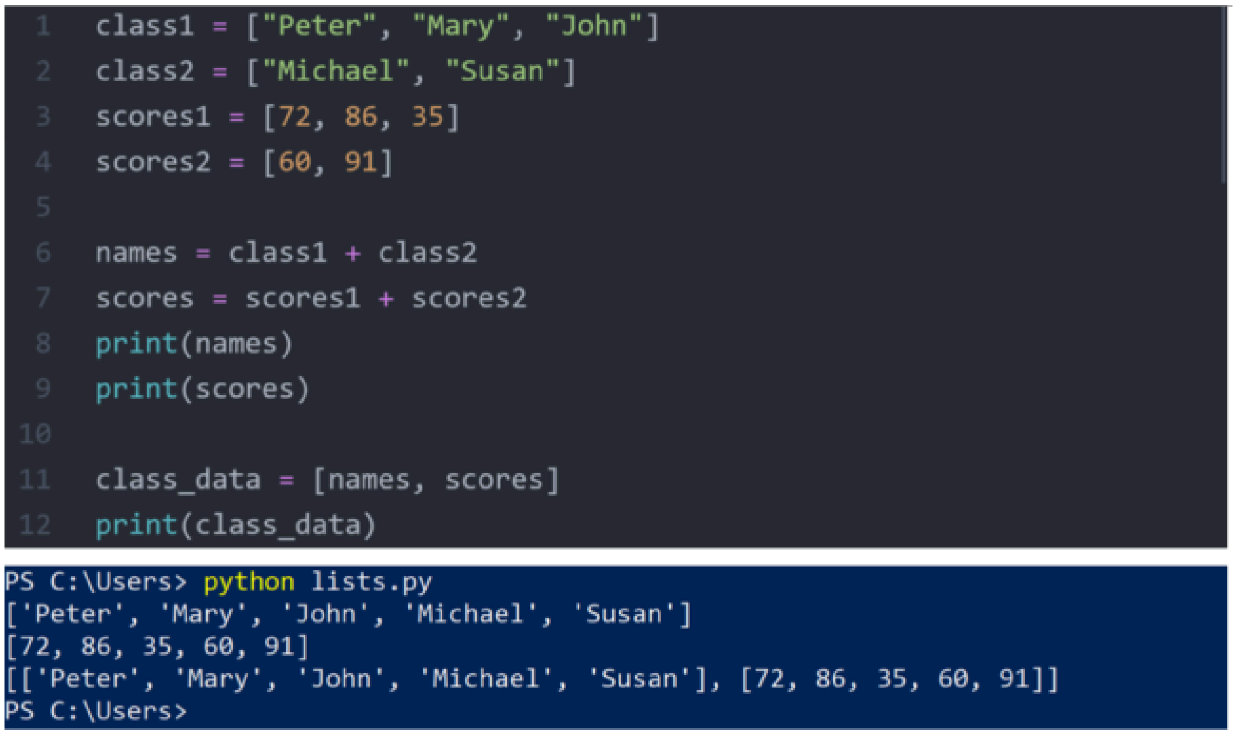
Another way to solve this problem is to merge two lists into a new list without combining the elements together. Instead, each element of the new list is a list and not a single value.

To merge two lists into a new list while keeping them as "list elements", we cannot use the addition operator as introduced before. Instead, we define the new list by putting the list names instead of some values as the elements.

```
list_name = [list1, list2, …]
```

The advantage of this merging technique is that we will always know that index 0 of the new list refers to the list `names` and index 1 to the list `scores` if the new list is a combination of two known lists in a fixed sequence. As a result, we will be able to trace back the origin and meaning of the data in the new list.

Example (Cont'd): Suppose we would like to put the two lists `names` and `scores` into a new list called `class_data` and keep them as lists in the new list.



```
class1 = ["Peter", "Mary", "John"]
class2 = ["Michael", "Susan"]
scores1 = [72, 86, 35]
scores2 = [60, 91]

names = class1 + class2
scores = scores1 + scores2
print(names)
print(scores)

class_data = [names, scores]
print(class_data)
```

```
PS C:\Users> python lists.py
['Peter', 'Mary', 'John', 'Michael', 'Susan']
[72, 86, 35, 60, 91]
[['Peter', 'Mary', 'John', 'Michael', 'Susan'], [72, 86, 35, 60, 91]]
PS C:\Users>
```

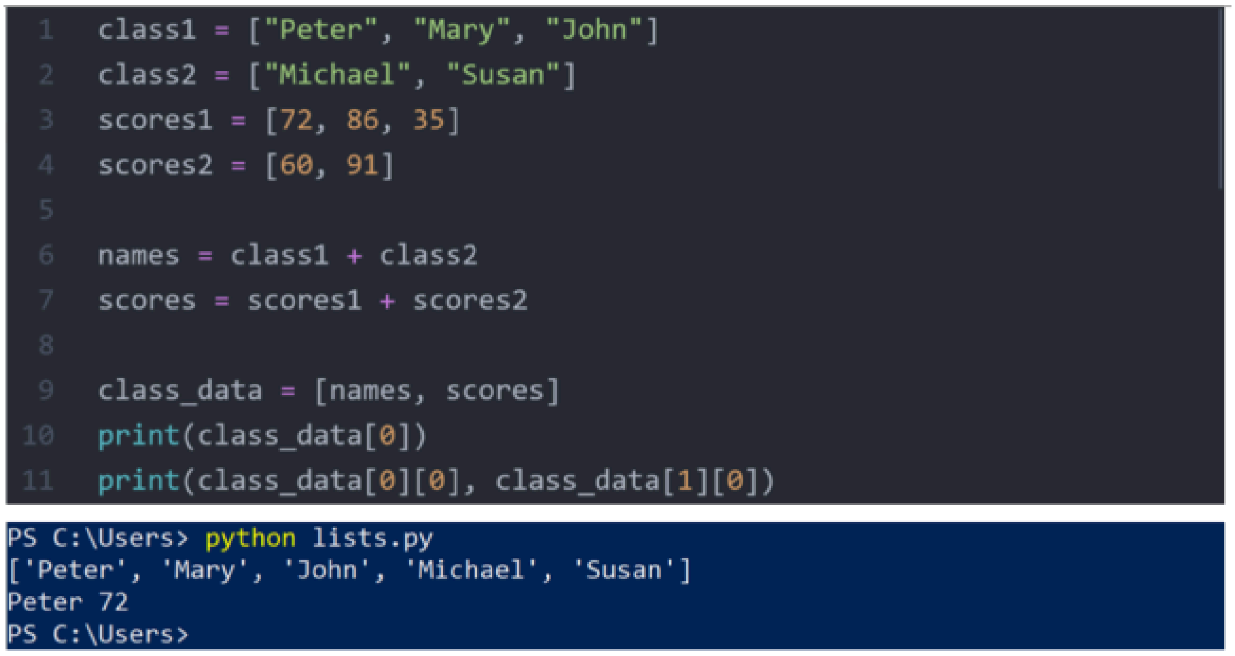
Figure 2.19 Merging Two Lists by Keeping Them as Lists in the New One

Different from the concatenation technique where all elements of the new list are just combined straightforwardly, we can see from Figure 2.19 that the elements of the list `names` are wrapped by each one pair of outer *and* inner square brackets to indicate that they belong to a list that in turn is an element of the new list `class_data`. The same can be observed for the elements of `scores` too.

Note that this merging technique is not limited to merging lists. We can also merge two tuples into a list and keep their types as tuple in the new list as well.

To access a single element in the merged list, we need to use the double index operator `[]` since the single index operator would return one of the original lists to us.

Example (Cont'd): Now we extract the first element as well as the first name and score from the merged list `class_data` and print them to the screen.



```
1   class1 = ["Peter", "Mary", "John"]
2   class2 = ["Michael", "Susan"]
3   scores1 = [72, 86, 35]
4   scores2 = [60, 91]
5
6   names = class1 + class2
7   scores = scores1 + scores2
8
9   class_data = [names, scores]
10  print(class_data[0])
11  print(class_data[0][0], class_data[1][0])
```

```
PS C:\Users> python lists.py
['Peter', 'Mary', 'John', 'Michael', 'Susan']
Peter 72
PS C:\Users>
```

Figure 2.20 Extracting Elements from a Merged List

In line 10, the object that we put in the `print()` function is `class_data[0]`, which is the first element of `class_data`. And this element happens to be the original list `names`. In line 11, we try to extract the first element from the first list within `class_data` by using the double index operator `class_data[0][0]`. As a result, Python extracts the first element in `class_data` first, which corresponds to the list with the `names` elements. From there, Python extracts the element with the index 0, which is "Peter" in this case. The same has also been carried out with `class_data[1][0]`, which is the first element of the second list. The resulting element is 72 here.

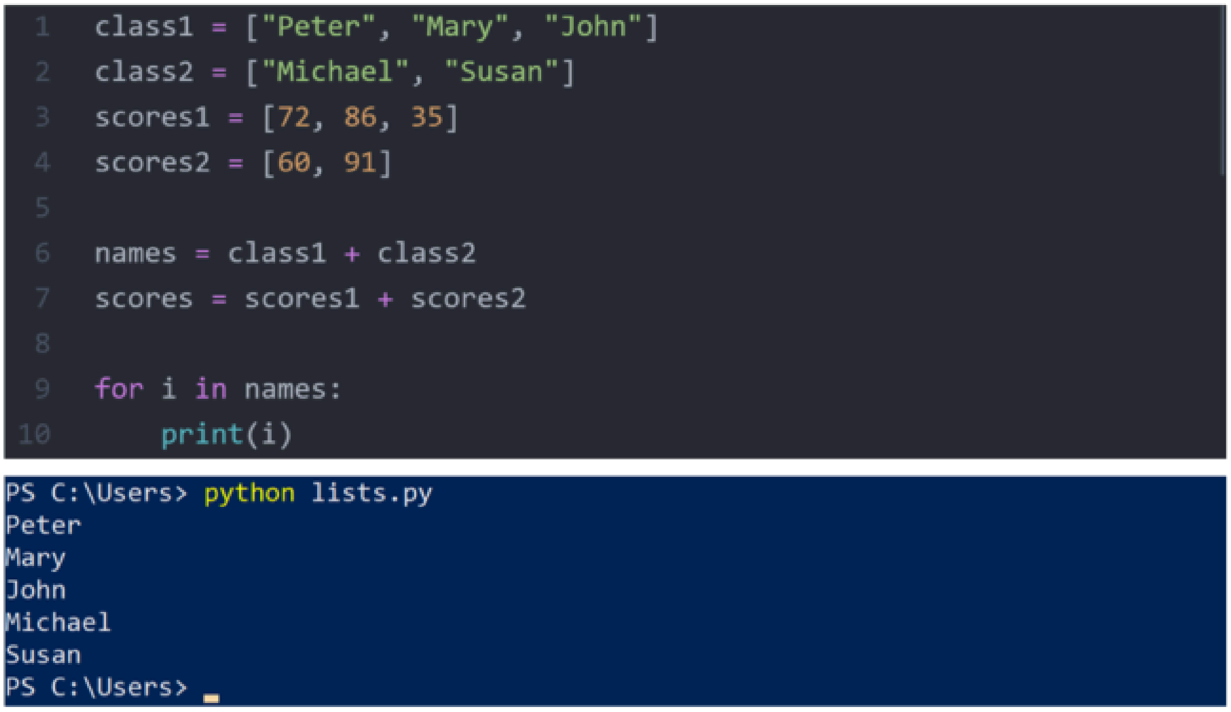
It is noteworthy that the indexing technique introduced in the above example also works for multiple indexing.

1.2.5 Printing Lists

In the previous study unit, we learned to use loops to carry out iterative tasks. In fact, loops can be very useful when working with lists. The reason is obvious: since the items of a list can be accessed by their indices, we can easily use loops to subset, print, and/or modify them.

Furthermore, we learned how to generate a list of integers to serve as sort of a counter for the iterations of the `for`-loops. Now, after being familiarised with the concept of lists, we do not always need the `range()` function to create these integers for us. Instead, we can simply use any available list as our counter. Nevertheless, the `range()` function can still be very useful in some situations.

Example (Cont'd): Suppose we would like to print all the student names of the two classes to the screen.



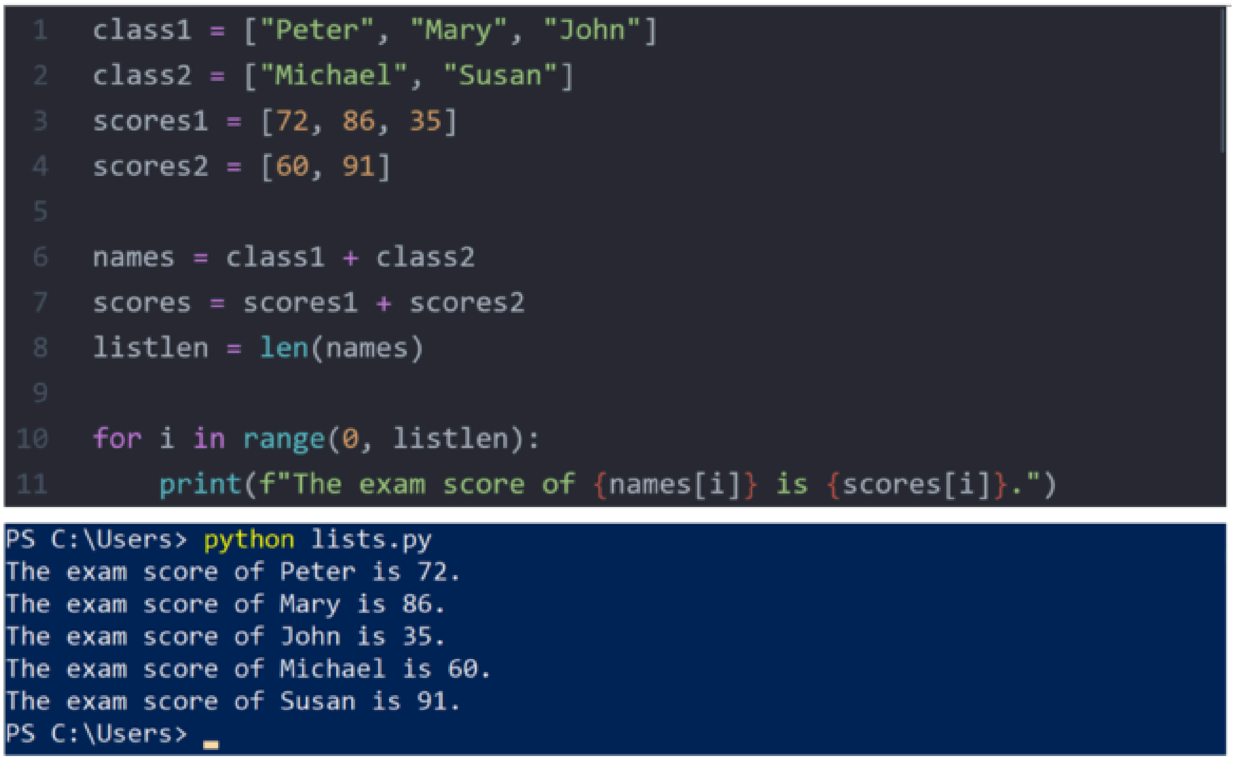
```
class1 = ["Peter", "Mary", "John"]
class2 = ["Michael", "Susan"]
scores1 = [72, 86, 35]
scores2 = [60, 91]

names = class1 + class2
scores = scores1 + scores2

for i in names:
    print(i)
```

```
PS C:\Users> python lists.py
Peter
Mary
John
Michael
Susan
PS C:\Users> _
```

Figure 2.21 Printing List Elements Line by Line to the Screen

In this example, we print the elements of the list `names` line by line to the screen. In each iteration, Python will assign the value of the current list element to the counter variable `i`, and the `print()` function will print the value stored in i to the screen.

Suppose we would like to modify the screen output and use string formatting to print the score and the student name in the same line by linking them up using more natural language.



```
class1 = ["Peter", "Mary", "John"]
class2 = ["Michael", "Susan"]
scores1 = [72, 86, 35]
scores2 = [60, 91]

names = class1 + class2
scores = scores1 + scores2
listlen = len(names)

for i in range(0, listlen):
    print(f"The exam score of {names[i]} is {scores[i]}.")
```

```
PS C:\Users> python lists.py
The exam score of Peter is 72.
The exam score of Mary is 86.
The exam score of John is 35.
The exam score of Michael is 60.
The exam score of Susan is 91.
PS C:\Users> _
```

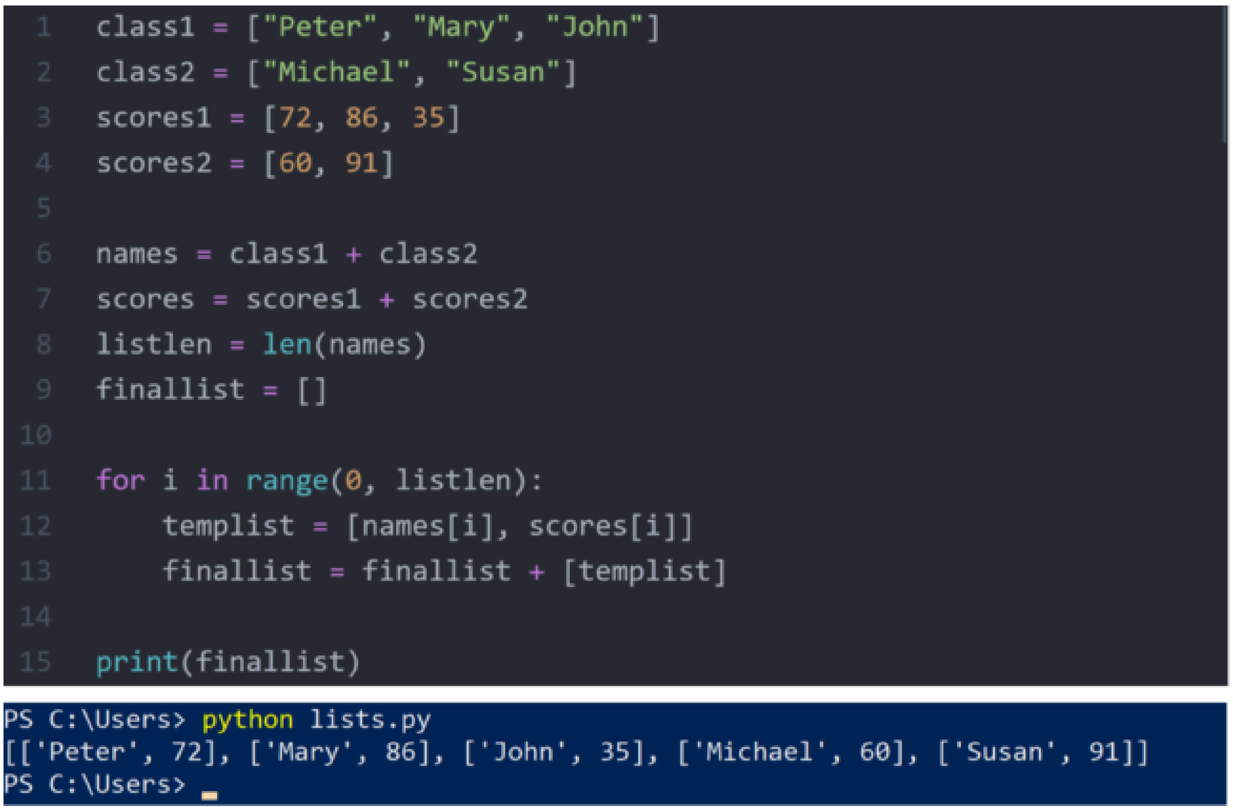
Figure 2.22 Printing Elements of Multiple Lists Using Formatted String

Since we want to run through two lists, `names` and `scores`, and print out their corresponding elements, we cannot use one of the lists as our counter in the `for`-loop. Thus, we need to use the `range()` function to generate a list of integers as our counter. Most conveniently, the integers should be exactly corresponding to the indices so that we can subset our lists within the `for`-loop directly. The obvious start index is 0, and the end index would be the number of items in our lists minus one. Since the `range()` function does not include the end value of the range in the result, we can therefore simply take the length of our list for this purpose. Here, we can apply the `len()` function, introduced in Chapter 1.1.4, to determine the length of `names`, and then store the result in the variable `listlen`, which will in turn be taken as the end of our integer list for the `range()` function.

Within the `for`-loop, we will have to instruct Python to print the element of `names` and `scores` with the index `i` for each iteration. And we can subset them by `names[i]` and `scores[i]`, respectively.

In the above example, we need to extract the information from two separate lists. In Chapter 1.2.4, we learned how to merge two lists into one single list where the original lists are kept as list elements in the new one, which makes the coding easier since we only need to work with one list. Instead of using the syntax we learned there, we can also use loops to carry out simple or complicated merging of lists.

Example (Cont'd): Suppose we would like to extract the first student's name and score from the two lists and put these data into a small list, then do the same for the second student, etc. Eventually, all these small lists will be merged into a single list.



```python
class1 = ["Peter", "Mary", "John"]
class2 = ["Michael", "Susan"]
scores1 = [72, 86, 35]
scores2 = [60, 91]

names = class1 + class2
scores = scores1 + scores2
listlen = len(names)
finallist = []

for i in range(0, listlen):
    templist = [names[i], scores[i]]
    finallist = finallist + [templist]

print(finallist)
```

```
PS C:\Users> python lists.py
[['Peter', 72], ['Mary', 86], ['John', 35], ['Michael', 60], ['Susan', 91]]
PS C:\Users> _
```

Figure 2.23 Rearranging Data Storage in Lists

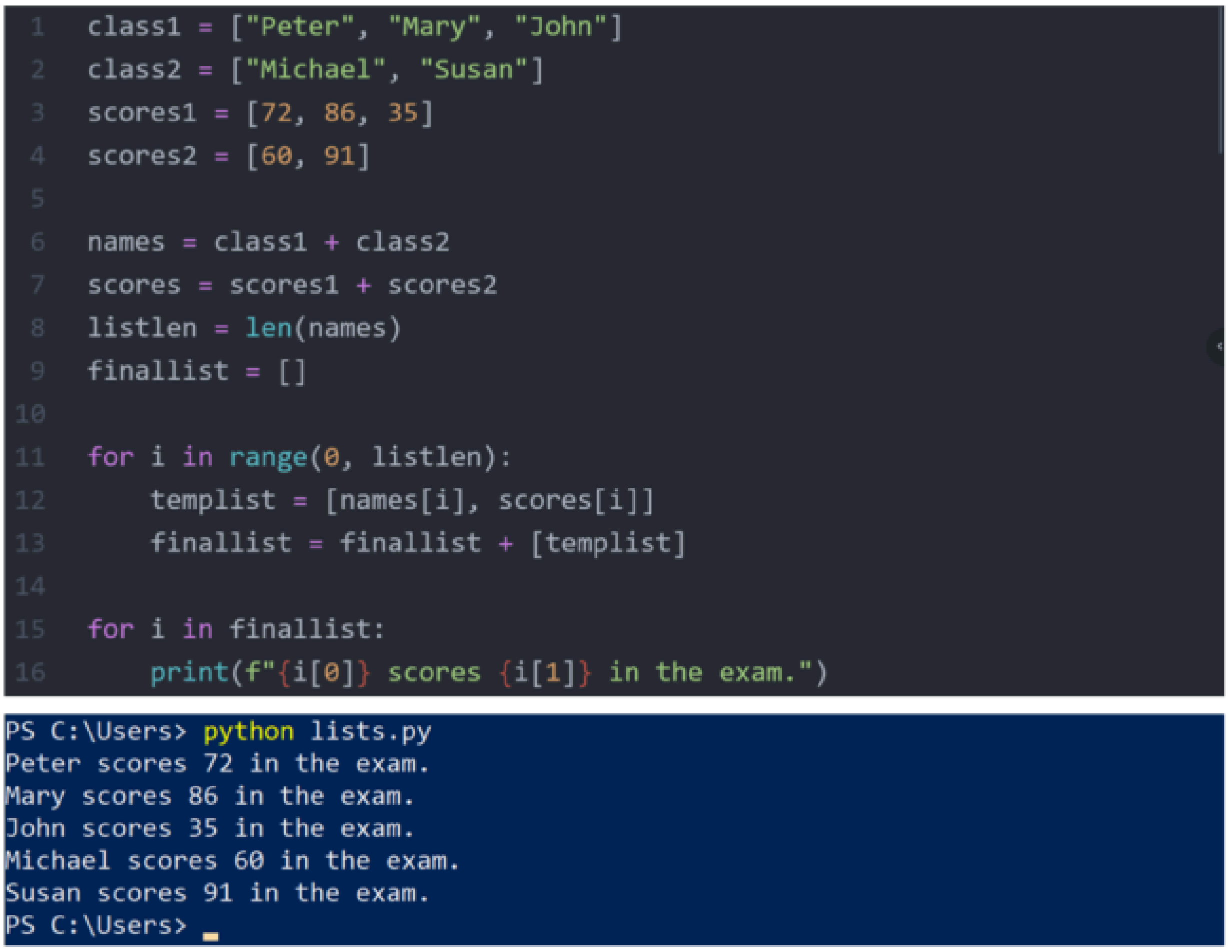
In line 12, we create a temporary list to store the pair of values that we have extracted from `names` and `scores`. This temporary list `templist` will then be appended to `finallist`, our target list, in line 13. This line is essential for the whole process as we would like to merge all the information into a single list eventually.

However, it is important that we define `finallist` as an empty list before the `for`-loop starts. To define an empty list, we simply assign a pair of square brackets `[]` with no content to our target list. If we did not define our target list first, line 13 would produce a syntax error as Python would not know what `finallist` is.

Furthermore, the instructions inside a loop are identical in every iteration unless we add an `if`-condition in it. If we defined `finallist` as an empty list inside the `for`-loop instead, the list would be re-initialised and become empty in each iteration. And if we simply have `finallist = templist` in line13, the complete content of `finallist` would be replaced in each iteration. The final output would then only consist of the data of the last student. It is therefore important to accumulate data in every iteration by appending `templist` to `finallist`.

The square brackets wrapping `templist` in line 13 is important since we want to keep the data of each student as a small "sublist" within `finallist`. If the square brackets are omitted, each element in the small list will be appended as an individual element to `finallist`. Hence, it is important to be familiarised with how brackets are placed in a syntax appropriately. The output can be quite different.

This arrangement of `finallist` is perhaps the most natural way to store the data. In each small list, we have the data of each student. In other words, if we extract an element of the list finalist, we will have all the data of the *one* corresponding student which can be quite convenient when dealing with these data further.



```
class1 = ["Peter", "Mary", "John"]
class2 = ["Michael", "Susan"]
scores1 = [72, 86, 35]
scores2 = [60, 91]

names = class1 + class2
scores = scores1 + scores2
listlen = len(names)
finallist = []

for i in range(0, listlen):
    templist = [names[i], scores[i]]
    finallist = finallist + [templist]

for i in finallist:
    print(f"{i[0]} scores {i[1]} in the exam.")
```

```
PS C:\Users> python lists.py
Peter scores 72 in the exam.
Mary scores 86 in the exam.
John scores 35 in the exam.
Michael scores 60 in the exam.
Susan scores 91 in the exam.
PS C:\Users> _
```

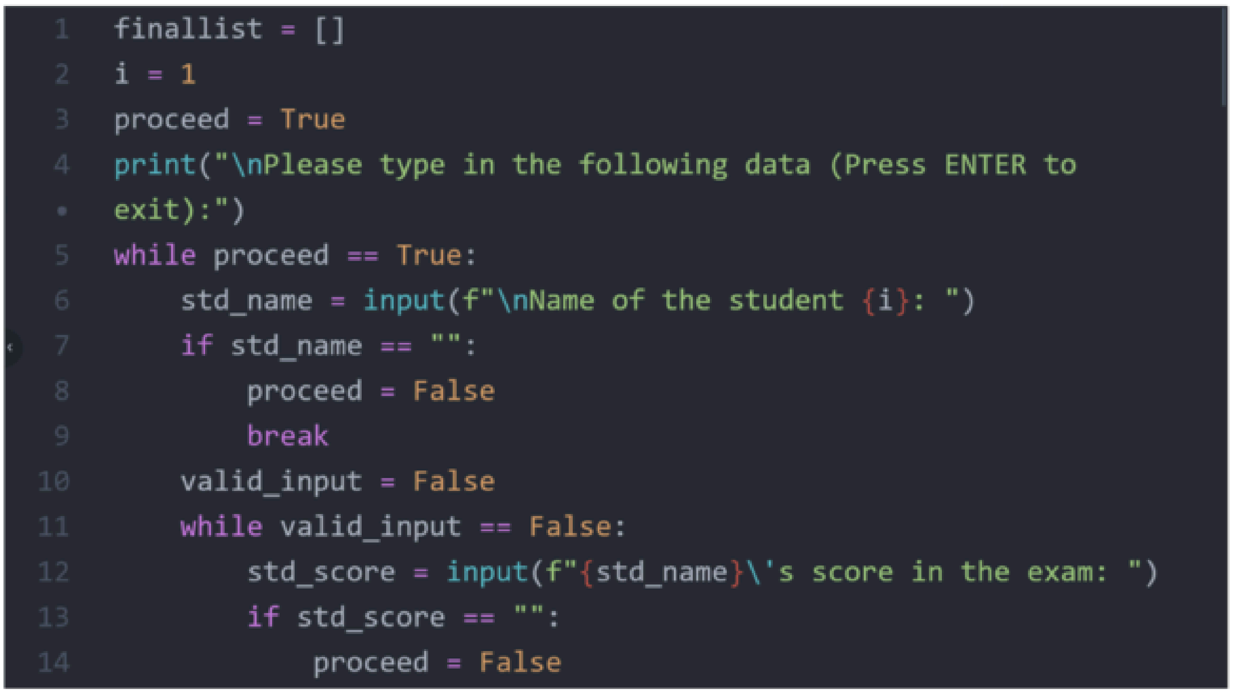
Figure 2.24 Printing Data of a List Using Formatted String

After creating `finallist`, we can use a `for`-loop to run through it. Within the `for`-loop, Python will extract one element from `finallist` in each iteration and store it in the variable `i`. Note that `i` is not a counter variable here. It is more like a temporary storage for the current element of the list that is being run through. Since each element of `finallist` is a list itself with 2 elements, we can subset them by `i[0]` and `i[1]`, respectively. From the construction concept of each of these small lists we know that the first element is the student's name, and the second element is the corresponding exam score. As a result, all we need to do is to put the subsets at the right place of the formatted string for screen output.

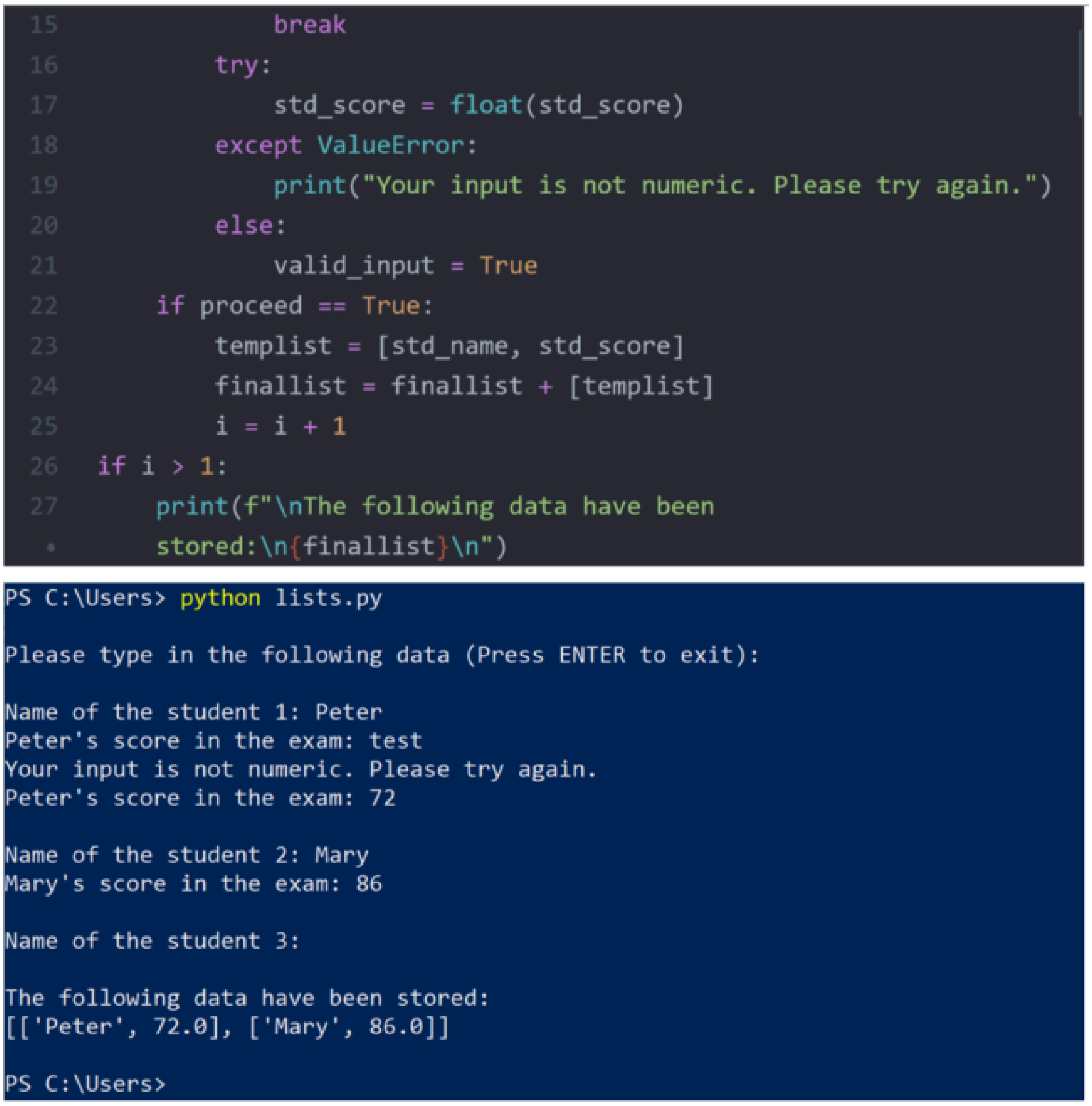
1.2.6 Entering Data to Lists

So far, we assign pre-defined values to the lists in our code directly. But we can also let the user enter his/her own data and store them into a list by the `input()` function. If we do not limit the number of entries like in Study Unit 1, a `while`-loop with an appropriate exit condition will be the right approach here.

Example (Cont'd): The user is now asked to enter the student's name and score, and he can stop entering by pressing ENTER for either the name or the score. After the entering process, the list will be printed to the screen for checking.



```python
finallist = []
i = 1
proceed = True
print("\nPlease type in the following data (Press ENTER to exit):")
while proceed == True:
    std_name = input(f"\nName of the student {i}: ")
    if std_name == "":
        proceed = False
        break
    valid_input = False
    while valid_input == False:
        std_score = input(f"{std_name}\'s score in the exam: ")
        if std_score == "":
            proceed = False
```



```
            break
        try:
            std_score = float(std_score)
        except ValueError:
            print("Your input is not numeric. Please try again.")
        else:
            valid_input = True
    if proceed == True:
        templist = [std_name, std_score]
        finallist = finallist + [templist]
        i = i + 1
if i > 1:
    print(f"\nThe following data have been stored:\n{finallist}\n")
```

```
PS C:\Users> python lists.py

Please type in the following data (Press ENTER to exit):

Name of the student 1: Peter
Peter's score in the exam: test
Your input is not numeric. Please try again.
Peter's score in the exam: 72

Name of the student 2: Mary
Mary's score in the exam: 86

Name of the student 3:

The following data have been stored:
[['Peter', 72.0], ['Mary', 86.0]]

PS C:\Users>
```

Figure 2.25 User's Input to a List

In this example, we use several elements that we have learned in this and the previous study units to provide a clear input environment to the user and the suitable functionality for data storage in a list.

- The program begins with the definition of an empty list called `finallist` that will be used to store the data eventually. This is the same step as in line 9 of Figure 2.23.

- We initiate two variables before the `while`-loop starts: the counter variable `i` which is set to 1 initially, and the Boolean variable `proceed` which controls whether the `while`-loop should continue with the next iteration or not.
- The `while`-loop will continue to run as long as `proceed` is `True`. Its value will only change from `True` to `False` if the user's input of either the name or the score is an empty string `""`, i.e., ENTER.
- We print an instruction to tell the user what to do before the loop starts. This instruction can also be put within the `while`-loop. In that case, the instruction will be printed to the screen in every iteration.
- We start the printing string in line 4 with an escape sequence "`\n`" to create a blank line between the system prompt and our instruction. It is visually more comfortable when the texts are not put too close together. We have built in several escape sequences for line break in the program to create blank lines for the same purpose.
- In line 6, we include the value of the counter variable in the formatted string to show the number of the student whose data the user is about to enter.
- The `if`-condition in line 7 checks whether the user's input for the student's name is an ENTER key or not. If it is the case, the program will switch `proceed` to `False` and break from the `while`-loop by the command `break`, which we learned in Chapter 5.3 of Study Unit 1. For the mechanism of breaking from the `while`-loop, the line `proceed = False` is insufficient since the remaining instructions within the `while`-loop will still be carried out although the user intents to stop the entering process immediately. As a result, we need to add the break-command to it. Nevertheless, for the sake of programming "cleanliness", we also add the line to change the value of `proceed` to `False`.
- Starting from line 10, we implement a `while`-loop controlled by a Boolean variable called `valid_input` to check whether the user's input of the score is numeric or not. This part of the code is mostly taken over from Chapter 5.4 in Study Unit 1. The main difference between the code there and our current one is that the user is allowed to press ENTER, i.e., an empty string to quit the

entering process instead of -1 as in Study Unit 1. As a result, we cannot convert the user's input to float or integer in the same line as the input() function, or else the `if`-condition in line 13 could become invalid.

- The exit mechanism for the input of the student's score is almost the same as the one for the student's name. The only difference here is that both syntaxes `proceed = False` and `break` are not redundant. The `break` command is used here to break out from the `while`-loop that controls the numeric input, and not the outer `while`-loop for entering data of multiple observations. The change of value for `proceed` from `True` to `False` prevents the merge of list in lines 22-25. Hence, the program will jump to the end of the `while`-loop and break from it due to `proceed = False`.
- The conversion of the variable `std_score` to a numeric value will be put in the `try`-block to prevent Python from stopping the program due to the occurrence of an error. If the input of `std_score` is indeed a number (but stored as string temporarily), the control variable `valid_input` will turn to `True` and the `while`-loop will stop iterating. However, if the input is a character string (with exception of an empty string), Python will print the warning message that is written in the `except`-block, and the `while`-loop will start a new iteration.
- In the formatted string of the `input()` function to enquire for the exam score in line 12, we embed the student's name that has just been entered by the user in line 6. This can prevent users from entering the score of another student due to the visual confusion caused by the mass of information on the screen.
- The same mechanism to store the data in `templist` and then to append them to `finallist` as in Figure 2.23 is employed here in line 23 and line 24. But the whole procedure will only be carried out if `proceed = True`. That is, if the inputs of both the name *and* the score are not empty strings.
- We increase our counter variable by 1 after all the data for the `i`th student have been typed in and confirmed by the user.
- Subsequent to the entire entering process, the program will print the entire list to the screen for checking purpose. However, this will only be executed if the

user has at least entered the entire set of data for *one* student. This control is implemented in case the user has already quitted the entering process during the first iteration and prevents us from printing an empty list. We have also built in several escape sequences for line breaks here to enhance the visual comfort of the output.

Read

Read the following official Python documentation: (https://docs.python.org/3/library/stdtypes.html#sequence-types-list-tuple-range) for details and examples on lists and on operations applicable to lists.

Read the following section of the textbook on looping over elements of lists:

Exercise 32 Loops and Lists

Read the following section of the textbook on accessing, adding, removing and joining elements of lists:

Exercise 34 Accessing Elements of Lists

Exercise 38 Doing Things to Lists

1.3 Dictionaries

Lesson Recording

Python Dictionaries

1.3.1 Defining & Extracting Dictionaries Values

Another useful built-in data type is the dictionary. It is best to think of a dictionary as an unordered set of `key:value` pairs, with the requirement that the keys are unique (within one dictionary). The `key:value` pairs are separated by commas and wrapped in a pair of braces.

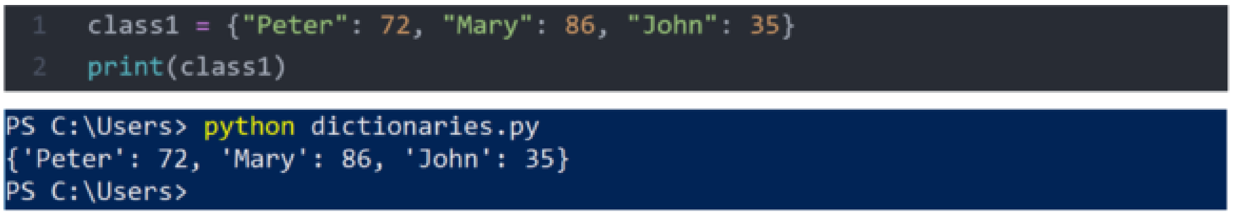
```
dictionary_name = {"key1":value1, "key2":value2, …}
```

Unlike lists and tuples, which are indexed by a range of numbers, dictionaries are indexed by keys, which are usually strings or numbers. As a result, we use keys to subset a dictionary instead of indices.

```
value = dictionary_name["key"]
```

Hence, we need to use the keys to extract values from the dictionary.

Example (Cont'd): Previously, we store the name and score of a student in a small list and merge all these "sub-lists" into a big list subsequently. Now, we save our data of `class1` in a dictionary instead. In the following, we will use the student names as the key and their scores as the value.

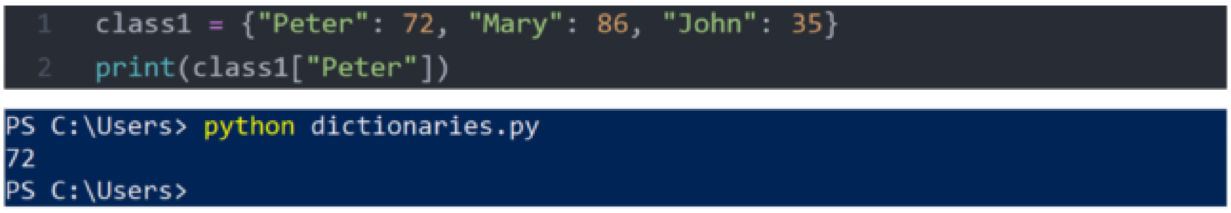


```
class1 = {"Peter": 72, "Mary": 86, "John": 35}
print(class1)
```

```
PS C:\Users> python dictionaries.py
{'Peter': 72, 'Mary': 86, 'John': 35}
PS C:\Users>
```

Figure 2.26 Defining a Dictionary

To extract the score of Peter, we can put the key, `"Peter"`, in the index operator `[]`.



```
class1 = {"Peter": 72, "Mary": 86, "John": 35}
print(class1["Peter"])
```

```
PS C:\Users> python dictionaries.py
72
PS C:\Users>
```

Figure 2.27 Extracting Value from a Dictionary

Note that if the key that we would like to refer to is a string, we will have to put it within a pair of quotation marks.

The advantage of dictionary is that we can choose a much suitable description for the value and use it as the key. The programmers or users do not need to rely on their memory or understanding of the code to trace back the nature and type of the values in a tuple or a list like our examples in Chapter 1.2.4 and Chapter 1.2.5.

1.3.2 Printing Dictionaries

We can use `for`-loops to print out the items from a dictionary line by line. However, the syntax is quite different from tuples and lists since each entry in a dictionary contains two elements: key and value. On the one hand, we can use the key to extract the corresponding value as demonstrated in Chapter 1.3.1, on the other hand, we cannot simply use the index operator `[]` to refer to the keys. One possibility to get the keys of a dictionary is to use the `.keys()` method:

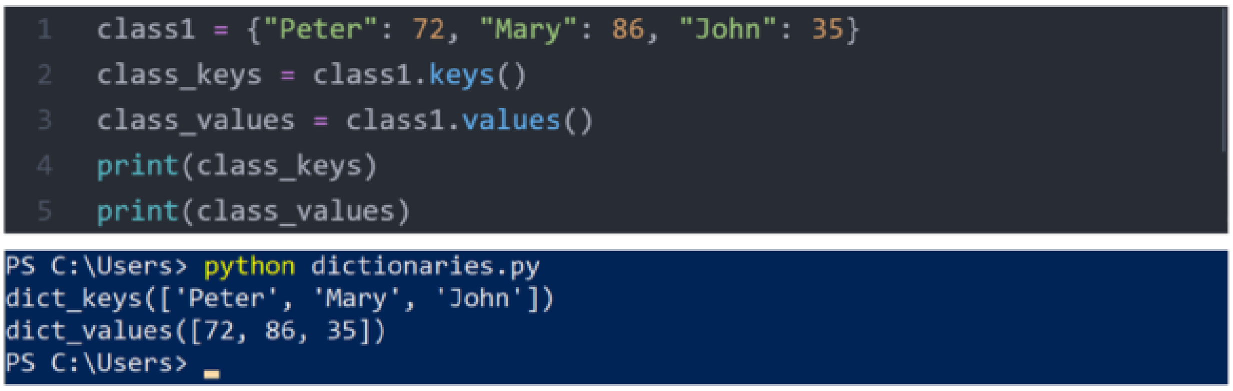
```
dictionary_name.keys()
```

Equivalently, the `.values()` method can extract all values from a dictionary.

```
dictionary_name.values()
```

Method is like a function to carry out certain actions on the object before the dot (`.`). We will have detailed discussion on methods and functions in later chapters. Here, the object is a dictionary, and the methods are `.keys()` and `.values()`. Applying these syntaxes on any dictionary, Python can extract all the keys and values from it.

Example (Cont'd): Suppose we would like to obtain all the keys and values separately from the dictionary `class1` defined in Figure 2.26.

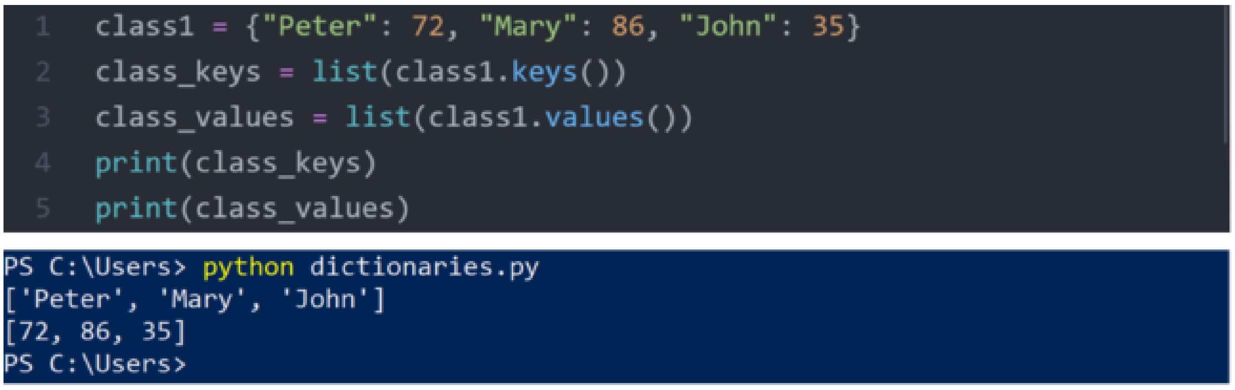


```
class1 = {"Peter": 72, "Mary": 86, "John": 35}
class_keys = class1.keys()
class_values = class1.values()
print(class_keys)
print(class_values)
```

```
PS C:\Users> python dictionaries.py
dict_keys(['Peter', 'Mary', 'John'])
dict_values([72, 86, 35])
PS C:\Users> _
```

Figure 2.28 Extracting Keys and Values from a Dictionary

The method `.keys()` returns an object called `dict_keys()` with the keys in our dictionary, and the method `.values()` returns an object called `dict_values()` with the values. However, we cannot work with these keys and values yet since we cannot extract them from the `dict_keys()` and `dict_values()` objects directly. We need to transform them to list by a function called `list()` first.

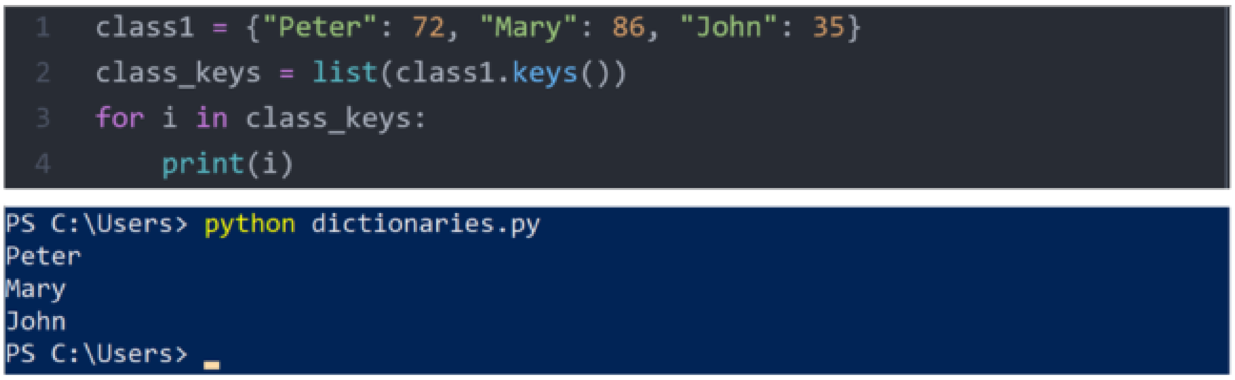


```
class1 = {"Peter": 72, "Mary": 86, "John": 35}
class_keys = list(class1.keys())
class_values = list(class1.values())
print(class_keys)
print(class_values)
```

```
PS C:\Users> python dictionaries.py
['Peter', 'Mary', 'John']
[72, 86, 35]
PS C:\Users>
```

Figure 2.29 Store Keys and Values from a Dictionary in Lists

The keys and values are now stored in their corresponding lists, recognisable by the square brackets around them. From the techniques introduced in Chapter 1.2, we can print the keys line by line now.

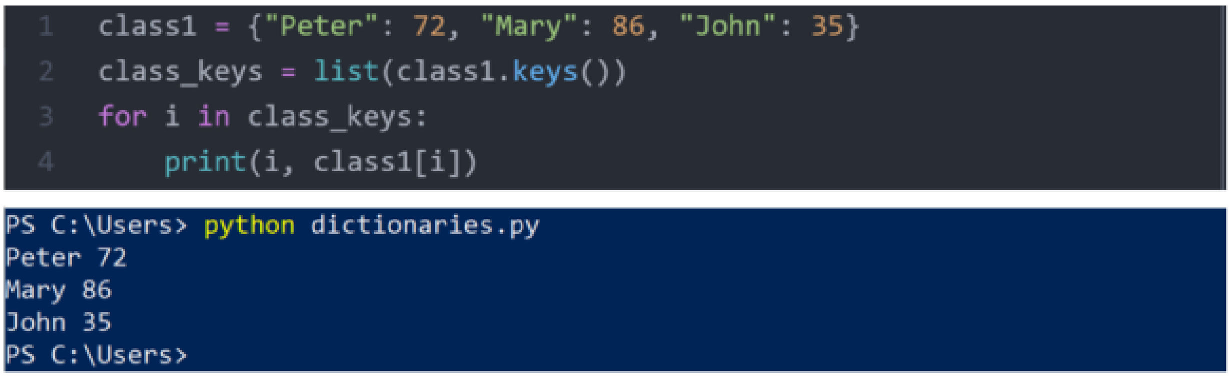


```python
class1 = {"Peter": 72, "Mary": 86, "John": 35}
class_keys = list(class1.keys())
for i in class_keys:
    print(i)
```

```
PS C:\Users> python dictionaries.py
Peter
Mary
John
PS C:\Users>
```

Figure 2.30 Print All the Keys from a Dictionary Line by Line

We can apply the same technique to print the values line by line too. Nevertheless, we can also use the index operator `[]` on the original list `class1` to print the keys and the corresponding values. To extract the keys individually, we need to initiate a `for`-loop that iterates through the list containing the dictionary keys, which is `class_keys` here. In each iteration within the `for`-loop, one key of the list will be stored in the variable `i`. The value can then be extracted by `class1[i]`.



```python
class1 = {"Peter": 72, "Mary": 86, "John": 35}
class_keys = list(class1.keys())
for i in class_keys:
    print(i, class1[i])
```

```
PS C:\Users> python dictionaries.py
Peter 72
Mary 86
John 35
PS C:\Users>
```

Figure 2.31 Print All the Keys and Values from a Dictionary Line by Line

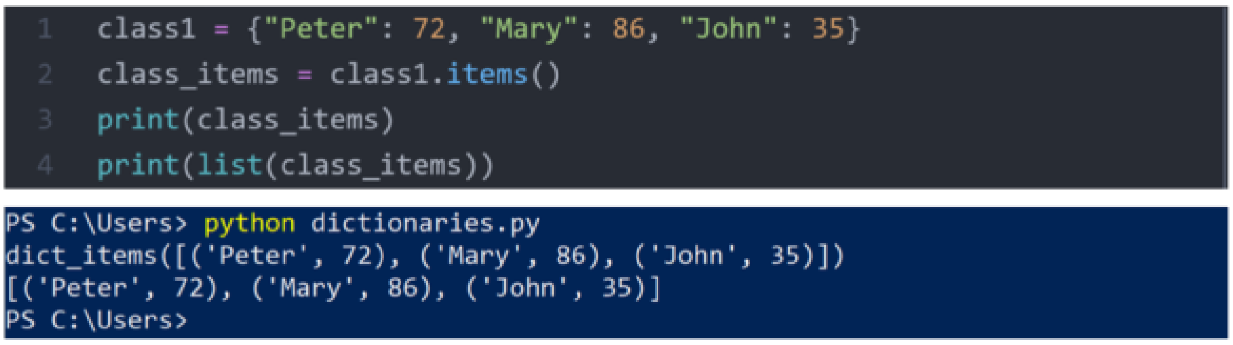
We put two objects separated by comma in the `print()` function. This is a command to tell Python to print these objects in the same line to the screen. And the values assigned in these objects are separated by an empty space between them.

If we want to extract all keys and values from a dictionary in the same step, we can apply the `.items()` method on a dictionary.

```
dictionary_name.items()
```

Same as the `.keys()` and `.values()` methods, the result returned by Python from the `.items()` method is not an object that can be accessed directly. Nevertheless, it can be converted to a list of keys and values by the `list()` function.

Example (Cont'd): We apply the `.items()` method on `class1` to obtain all the keys and values of the dictionary.



```
class1 = {"Peter": 72, "Mary": 86, "John": 35}
class_items = class1.items()
print(class_items)
print(list(class_items))
```

```
PS C:\Users> python dictionaries.py
dict_items([('Peter', 72), ('Mary', 86), ('John', 35)])
[('Peter', 72), ('Mary', 86), ('John', 35)]
PS C:\Users>
```

Figure 2.32 Extracting Items from a Dictionary

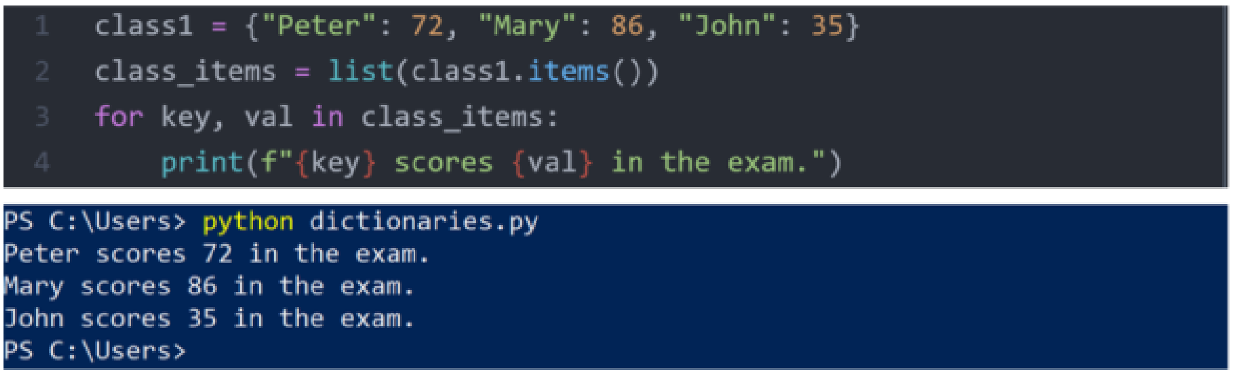
The first `print()` function prints the `dict_items()` object returned from the `.items()` method to the screen. This object contains every pair of keys and values from the dictionary stored in a tuple. In the second `print()` function, we convert the `dict_items()` object to a list. And the printed object is a list with each pair of keys and values stored in a tuple as its elements.

The list created from a `dict_items()` object contains tuples that have two elements in each of them: the keys and the values. We can easily use a `for`-loop and the index operator to print the contents. Here, we introduce an extension of the `for`-loop so that we can omit the indices when referring to the keys and values.

```
for element1, element2 in list(dictionary_name.items()):
    instructions
```

The syntax `list(dictionary_name.items())` in the above `for`-loop can certainly be replaced by any defined list created from a dictionary. The main difference between the usual `for`-loop we know and the one we introduce here is that we use two temporary storage variables instead of one in it. The mechanism is rather simple. In each iteration, Python will extract one tuple from the list, and each element of the tuple will then be stored in one of these storage variables. Since the keys are stored first in the tuples, `element1` will contain the key, and `element2` will contain the value. If we want to use these two values in our instructions, we can simply refer to these variables without using the index operator.

Example (Cont'd): Now we use the double storage variables in a `for`-loop to print out the keys and values of our `dict_items()` object after being converted to a list.



```
class1 = {"Peter": 72, "Mary": 86, "John": 35}
class_items = list(class1.items())
for key, val in class_items:
    print(f"{key} scores {val} in the exam.")
```

```
PS C:\Users> python dictionaries.py
Peter scores 72 in the exam.
Mary scores 86 in the exam.
John scores 35 in the exam.
PS C:\Users>
```

Figure 2.33 Print Items from a Dictionary after Converting it to a List

In line 2, we extract the `dict_items()` object and convert it directly to a list named `class_items`. The two storage variables of the `for`-loops are `key` and `val` here. And in the `print()` function, we use a formatted string to print out the two values in a normal sentence for the user to read.

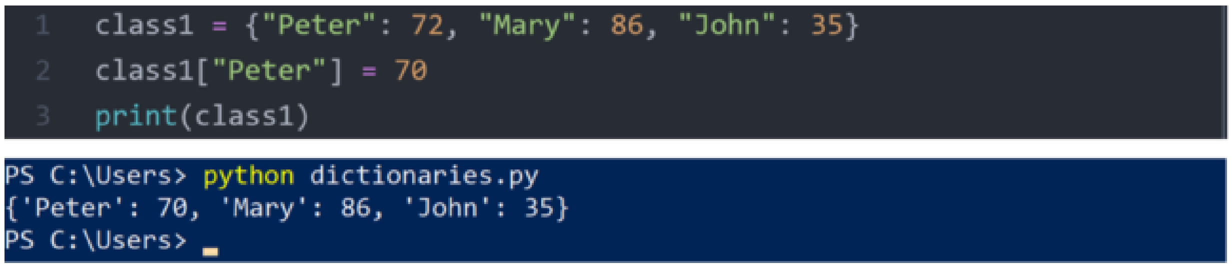
1.3.3 Editing Dictionaries

We can change a value of a dictionary by assigning a new value to a certain key.

```
dictionary_name["key"] = value
```

The value can be a numeric value, a character string, a tuple, or a list.

Example (Cont'd): Suppose the score of Peter was 70 instead of 72. We can simply change it by assigning a new value to the key "Peter".

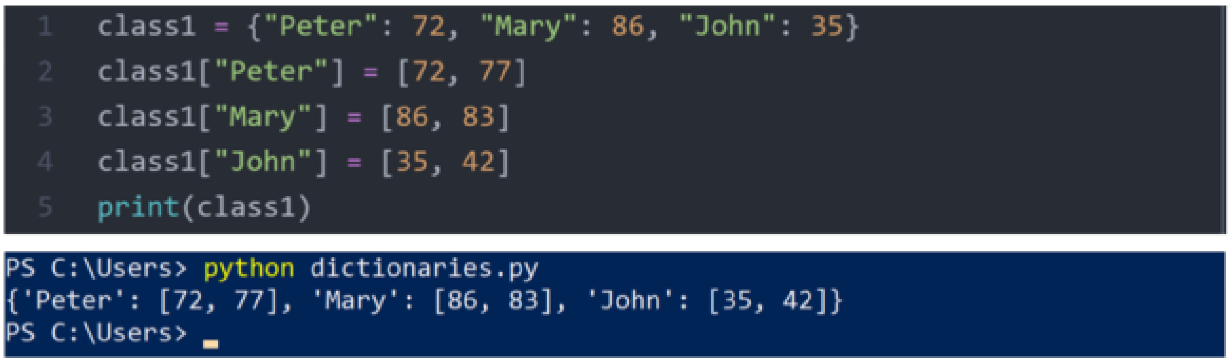


```
class1 = {"Peter": 72, "Mary": 86, "John": 35}
class1["Peter"] = 70
print(class1)
```

```
PS C:\Users> python dictionaries.py
{'Peter': 70, 'Mary': 86, 'John': 35}
PS C:\Users> _
```

Figure 2.34 Change One Value in a Dictionary

There was then a second exam and each of our three students has a new score in addition to the scores that are already stored in the dictionary. Below is one rather trivial approach to carry out this task.

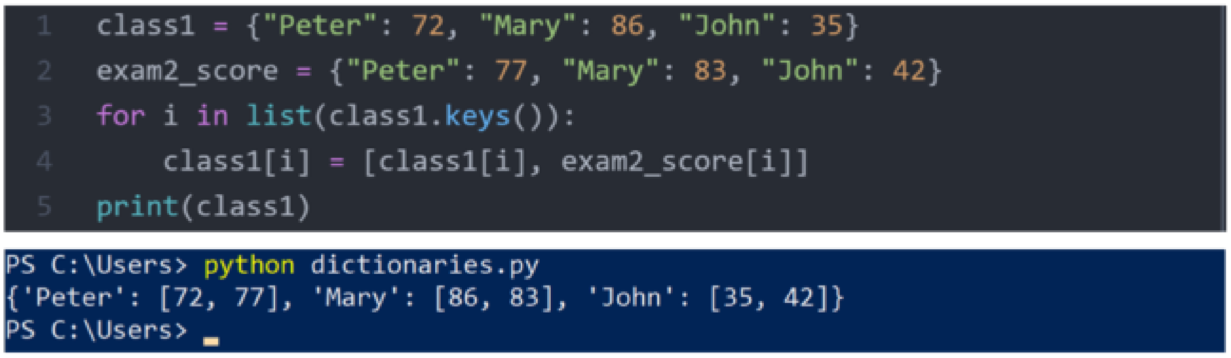


```
class1 = {"Peter": 72, "Mary": 86, "John": 35}
class1["Peter"] = [72, 77]
class1["Mary"] = [86, 83]
class1["John"] = [35, 42]
print(class1)
```

```
PS C:\Users> python dictionaries.py
{'Peter': [72, 77], 'Mary': [86, 83], 'John': [35, 42]}
PS C:\Users> _
```

Figure 2.35 Add Items to Every Dictionary Key and Convert Values to Lists

The above approach replaces the original values in the dictionary by some lists. Each list contains two scores of each student. Therefore, we must include the first score of each student in the list as well, which is not quite elegant in terms of programming conciseness. Another possibility is to use a `for`-loop to convert the value of each key and append the new score to it.



```
class1 = {"Peter": 72, "Mary": 86, "John": 35}
exam2_score = {"Peter": 77, "Mary": 83, "John": 42}
for i in list(class1.keys()):
    class1[i] = [class1[i], exam2_score[i]]
print(class1)
```

```
PS C:\Users> python dictionaries.py
{'Peter': [72, 77], 'Mary': [86, 83], 'John': [35, 42]}
PS C:\Users> _
```

Figure 2.36 Print Items from a Dictionary after Converting to Lists Using `for`-loops

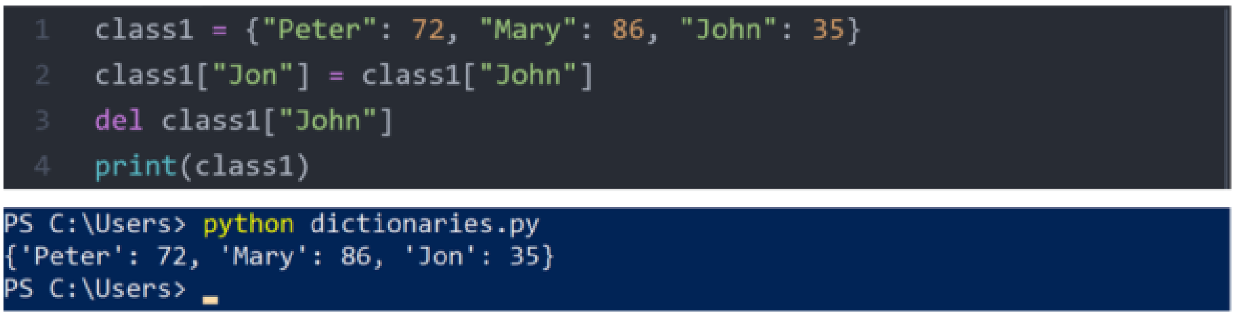
In this approach, we defined a new dictionary named `exam2_score` to store the scores of the second exam. We also use the students' names as our dictionary key here. Hence, `class1` and `exam2_score` share the same keys which makes the mechanism within the `for`-loops much easier to handle. The list that is used for the `for`-loop to iterate is the key list of the dictionary `class1`. In each iteration, the variable `i` will store one key from the list. Since both dictionaries have identical keys, we can use the dictionary subsetting technique on both dictionaries by putting `i` in the index operator. Subsequently, we merge the extracted values from both dictionaries to one list, and then assign it to the corresponding key of `class1`. Eventually, we print out the edited dictionary for checking purpose.

While assigning new values to a key in a dictionary is rather straightforward, editing a key in a dictionary is not a simple task in Python. Basically, the keys of a dictionary are immutable and cannot be changed directly. But we can create a new key in a dictionary, take over the values from the old key, and then delete the old key in the last step. To delete a key in a dictionary, we can use the syntax `del`.

```
del object
```

Note that the object in this syntax does not only refer to a key in a dictionary. It can be any Python object such as a variable, a list, a tuple, a dictionary, etc. Once an object is deleted, it will no longer be available in the program.

Example (Cont'd): Suppose the key "John" was a typo when entering the data. It should be "Jon" instead.



```
class1 = {"Peter": 72, "Mary": 86, "John": 35}
class1["Jon"] = class1["John"]
del class1["John"]
print(class1)
```

```
PS C:\Users> python dictionaries.py
{'Peter': 72, 'Mary': 86, 'Jon': 35}
PS C:\Users> _
```

Figure 2.37 Changing a key of a Dictionary

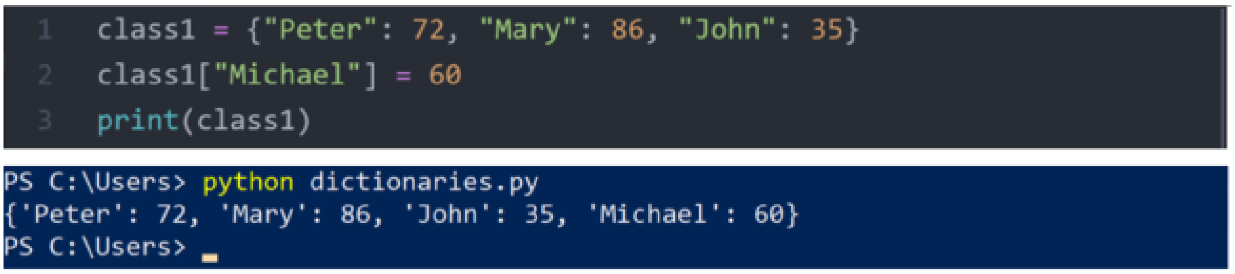
First, we assign the value of "John" in the dictionary `class1` to a new key called "Jon" of the same dictionary. Subsequently, we delete the key "John" and its value as an entire object from the running program.

On the other hand, if we want to add a new key to a dictionary, we can simply assign a value to a new dictionary key.

```
dictionary["new key"] = value
```

Basically, the syntax to add a new key is just the same as the syntax to edit an existing key. The only difference is that the key put in the index operator must be a new one if we wish to add a new item.

Example (Cont'd): Suppose we would like to add a new student, Michael, who scores 60 in the exam, to our dictionary.



```
class1 = {"Peter": 72, "Mary": 86, "John": 35}
class1["Michael"] = 60
print(class1)
```

```
PS C:\Users> python dictionaries.py
{'Peter': 72, 'Mary': 86, 'John': 35, 'Michael': 60}
PS C:\Users> _
```

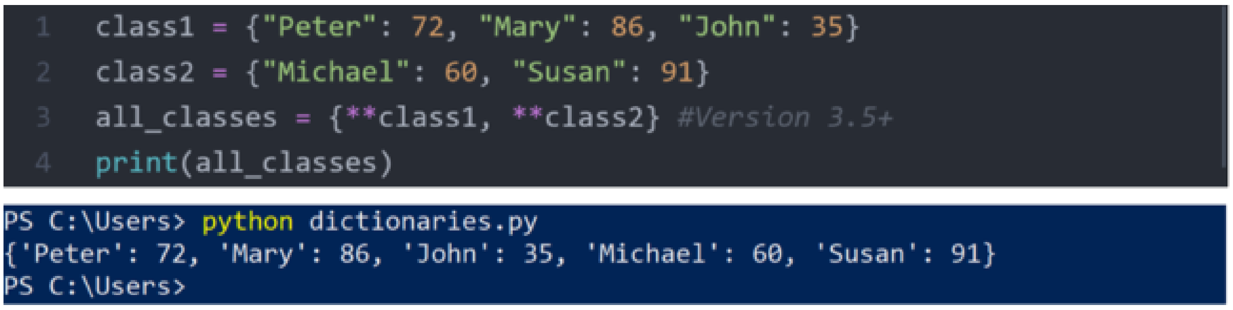
Figure 2.38 Adding a New Key to a Dictionary

However, if we want to merge two dictionaries, Python offers two options.

```
new_dictionary = dictionary1 | dictionary2 #Version 3.9+
new_dictionary = {**dictionary1, **dictionary2} #Version
 3.5+
```

The first syntax is a new option available from Python version 3.9 onwards (Recall from Study Unit 1 that this study guide is written based on Python version 3.9). It uses the "Bitwise `Or`" operator, which is a vertical line "`|`", to merge two dictionaries. The second syntax is an option available from Python version 3.5 onwards.

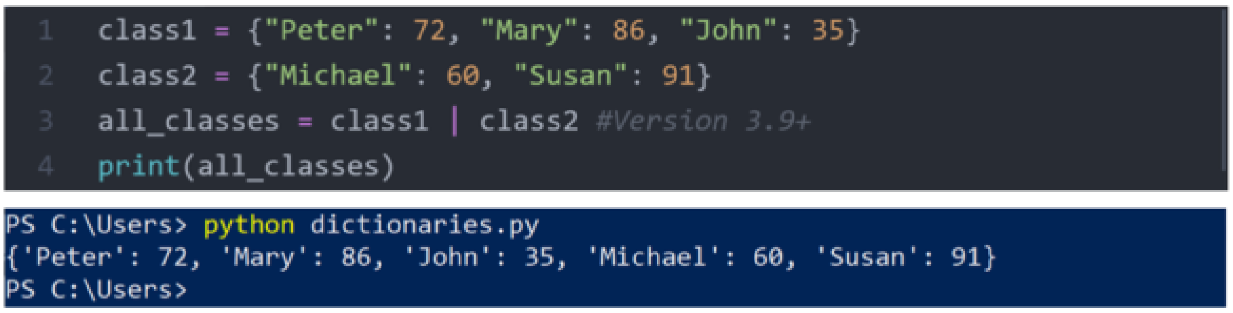
Example (Cont'd): Suppose we have a second dictionary with the exam results of a second class, and we would like to merge these two dictionaries together.



```
class1 = {"Peter": 72, "Mary": 86, "John": 35}
class2 = {"Michael": 60, "Susan": 91}
all_classes = {**class1, **class2} #Version 3.5+
print(all_classes)
```

```
PS C:\Users> python dictionaries.py
{'Peter': 72, 'Mary': 86, 'John': 35, 'Michael': 60, 'Susan': 91}
PS C:\Users>
```

Figure 2.39 Merging Two Dictionaries Using the v3.5+ Option



```
class1 = {"Peter": 72, "Mary": 86, "John": 35}
class2 = {"Michael": 60, "Susan": 91}
all_classes = class1 | class2 #Version 3.9+
print(all_classes)
```

```
PS C:\Users> python dictionaries.py
{'Peter': 72, 'Mary': 86, 'John': 35, 'Michael': 60, 'Susan': 91}
PS C:\Users>
```

Figure 2.40 Merging Two Dictionaries Using the v3.9+ Option

As we can see, the two methods deliver the same result.

Read

Read the following section of the textbook on creating and manipulating dictionaries:

Exercise 39 Dictionaries, Oh Lovely Dictionaries

Read the following official Python documentation for more details and examples on dictionaries:

https://docs.python.org/3/library/stdtypes.html#typesmapping

Chapter 2: Integrated Methods and Functions

Lesson Recording

Integrated Methods and Functions in Python

2.1 Built-In Functions in Python

In the previous chapters, we have come across various built-in functions of Python, such as `print()`, `input()`, `int()`, etc. These functions are always available in the Python environment. A function is a routine program that processes values which are passed on to it as an argument, or a parameter, and returns some results to the user eventually. For example, a float value is passed on to the `int()` function as an argument, Python then removes all the decimal digits and returns an integer as result.

The following table contains all the built-in functions in alphabetical order.

Table 2.1 Built-in Functions of Python

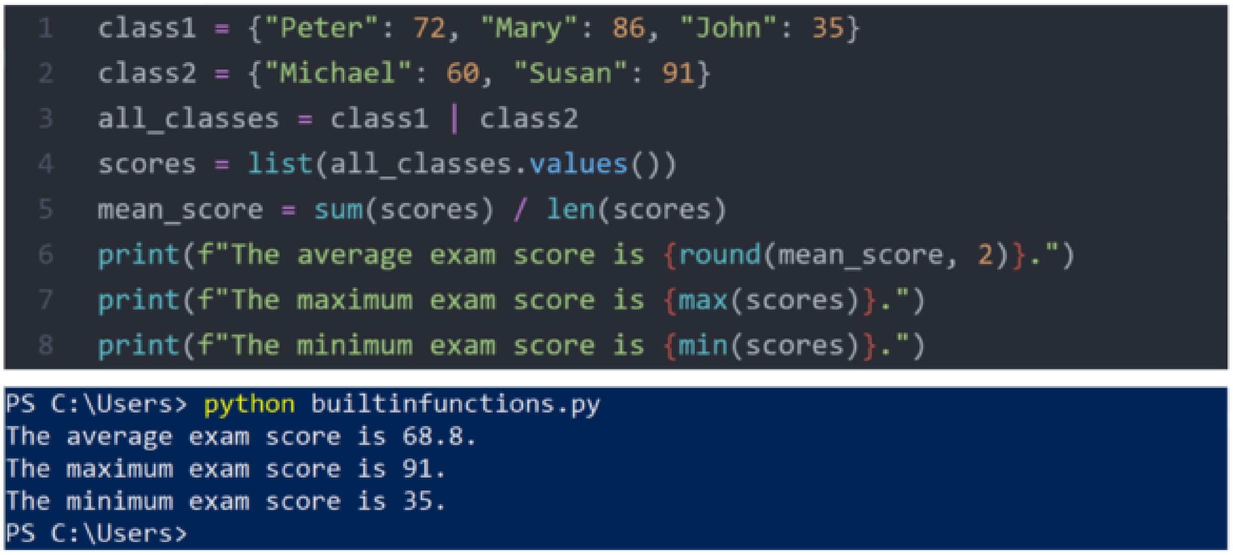
`abs()`	`all()`	`any()`	`ascii()`
`bin()`	`bool()`	`breakpoint()`	`bytearray()`
`bytes()`	`callable()`	`chr()`	`classmethod()`
`compile()`	`complex()`	`delattr()`	`dict()`
`dir()`	`divmod()`	`enumerate()`	`eval()`
`exec()`	`filter()`	`float()`	`format()`

`frozenset()`	`getattr()`	`globals()`	`hasattr()`
`hash()`	`help()`	`hex()`	`id()`
`input()`	`int()`	`isinstance()`	`issubclass()`
`iter()`	`len()`	`list()`	`locals()`
`map()`	`max()`	`memoryview()`	`min()`
`next()`	`object()`	`oct()`	`open()`
`ord()`	`pow()`	`print()`	`property()`
`range()`	`repr()`	`reversed()`	`round()`
`set()`	`setattr()`	`slice()`	`sorted()`
`staticmethod()`	`str()`	`sum()`	`super()`
`tuple()`	`type()`	`vars()`	`zip()`
`__import__()`			

(Source: https://docs.python.org/3/library/functions.html)

Some of the listed functions are rather straightforward such as `abs()`, `sum()`, `round()`, etc. There are also some that are quite unclear in terms of their functionality or area of use such as `frozenset()` or `staticmethod()` just by looking at their names. You can visit the website https://docs.python.org/3/library/functions.html to get detailed explanation on how to integrate and apply all these functions in Python programs.

Example (Cont'd): Suppose we would like to summarise the combined exam results of `class1` and `class2` by calculating their mean, maximum and minimum.



```python
1   class1 = {"Peter": 72, "Mary": 86, "John": 35}
2   class2 = {"Michael": 60, "Susan": 91}
3   all_classes = class1 | class2
4   scores = list(all_classes.values())
5   mean_score = sum(scores) / len(scores)
6   print(f"The average exam score is {round(mean_score, 2)}.")
7   print(f"The maximum exam score is {max(scores)}.")
8   print(f"The minimum exam score is {min(scores)}.")
```

```
PS C:\Users> python builtinfunctions.py
The average exam score is 68.8.
The maximum exam score is 91.
The minimum exam score is 35.
PS C:\Users>
```

Figure 2.41 Computing Statistics of a Dictionary's Values

In the first three lines, we repeat the syntaxes from Chapter 1.3.2 to merge two dictionaries into a new one called `all_classes`. Subsequently, all the exam scores are extracted by the `.values()` method from `all_classes`. In line 5, we use the `sum()` function to add up all the numbers stored in `scores` and divide the result by the number of elements in `scores`, determined by the `len()` function, to obtain the mean exam scores. The mean will then be rounded to two decimal places by the `round()` function when embedding it in a formatted string for printing purpose. In the last two lines, we determine the highest and lowest exam scores by the `max()` and `min()` functions and include them in the formatted strings for the `print()` function to print them onto the screen.

You may wonder why some rather basic functions such as a function for the calculation of the mean is missing in the above list. Some of those functions may be included in some common packages that will be introduced in Chapter 4. Some of them could be built-in methods instead, which will be introduced in the next section.

Read

Read the following official Python documentation for more details and examples on Python functions:

https://docs.python.org/3/library/functions.html

2.2 Built-In Methods in Python

Some routines in Python are not supposed to be applied as functions in the Python environment; instead, they are methods that can be applied to the objects they are attached to. In Study Unit 1, we learned the method `.format()` for format printing; and in this study unit, we come across the .keys(), **.values()** and **.items()** methods that can be applied on dictionaries to extract their keys and values. Same as functions, there are built-in methods that are always available in the Python environment. These methods will return results to the program once they are applied on defined objects during runtime. However, each method can only be applied to a certain object type. Below is a list of selected built-in methods of Python.

Table 2.2 Built-in Methods of Python

Method	Description	Applicable Object Type
`append()`	Adds an element at the end of the list	List
`capitalize()`	Converts the first character to upper case	String
`casefold()`	Converts string into lower case	String

Method	Description	Applicable Object Type
`center()`	Returns a centred string	String
`clear()`	Removes all the elements	List, Dictionary
`copy()`	Returns a copy	List, Dictionary
`count()`	Returns the number of times a specified value occurs in a string	String
`count()`	Returns the number of elements with the specified value	List, Tuple
`endswith()`	Returns `True` if the string ends with the specified value	String
`extend()`	Add the elements of a list (or any iterable), to the end of the current list	List
`find()`	Searches the string for a specified value and returns the position of where it was found	String
`fromkeys()`	Returns a dictionary with the specified keys and value	Dictionary

Method	Description	Applicable Object Type
`get()`	Returns the value of the specified key	Dictionary
`index()`	Returns the position of the first element with the specified value	List, Tuple
`insert()`	Adds an element at the specified position	List
`isalnum()`	Returns `True` if all characters in the string are alphanumeric	String
`isalpha()`	Returns `True` if all characters in the string are in the alphabet	String
`islower()`	Returns `True` if all characters in the string are lower case	String
`isnumeric()`	Returns `True` if all characters in the string are numeric	String
`isspace()`	Returns `True` if all characters in the string are whitespaces	String

Method	Description	Applicable Object Type
`istitle()`	Returns `True` if the string follows the rules of a title	String
`isupper()`	Returns `True` if all characters in the string are upper case	String
`items()`	Returns a list containing a tuple for each key value pair	Dictionary
`join()`	Joins the elements of an iterable to the end of the string	String
`keys()`	Returns a list containing the dictionary's keys	Dictionary
`lower()`	Converts a string into lower case	String
`lstrip()`	Returns a left trim version of the string	String
`pop()`	Removes the element at the specified position	List, Dictionary
`popitem()`	Removes the last inserted key-value pair	Dictionary

Method	**Description**	**Applicable Object Type**
`remove()`	Removes the first item with the specified value	List
`replace()`	Returns a string where a specified value is replaced with a specified value	String
`reverse()`	Reverses the order of the list	List
`rfind()`	Searches the string for a specified value and returns the last position of where it was found	String
`rstrip()`	Returns a right trim version of the string	String
`sort()`	Sorts the list	List
`split()`	Splits the string at the specified separator, and returns a list	String
`splitlines()`	Splits the string at line breaks and returns a list	String
`startswith()`	Returns true if the string starts with the specified value	String

Method	Description	Applicable Object Type
`strip()`	Returns a trimmed version of the string	String
`swapcase()`	Swaps cases, lower case becomes upper case and vice versa	String
`title()`	Converts the first character of each word to upper case	String
`update()`	Updates the dictionary with the specified key-value pairs	Dictionary
`upper()`	Converts a string into upper case	String
`values()`	Returns a list of all the values in the dictionary	Dictionary
`zfill()`	Fills the string with a specified number of 0 values at the beginning	String

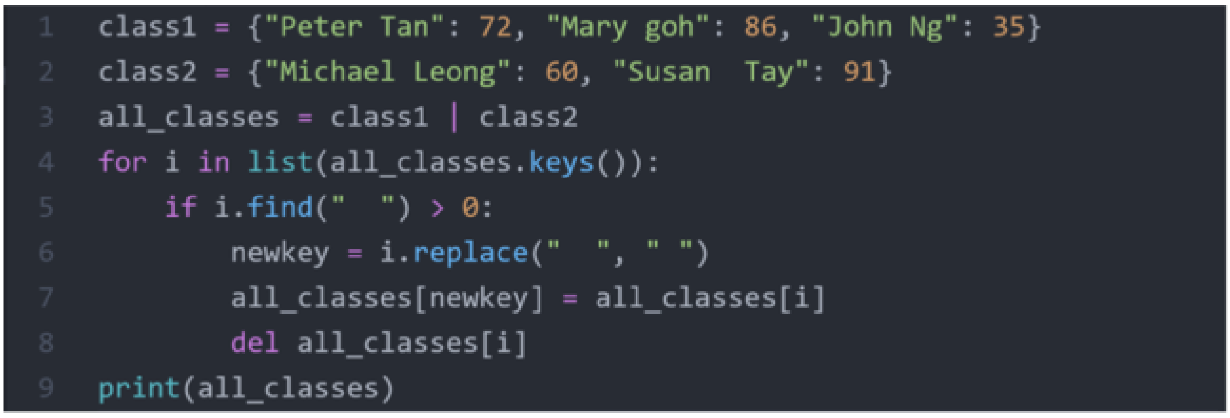
(Source: https://www.w3schools.com/python/default.asp)

The complete list of methods can be found on:

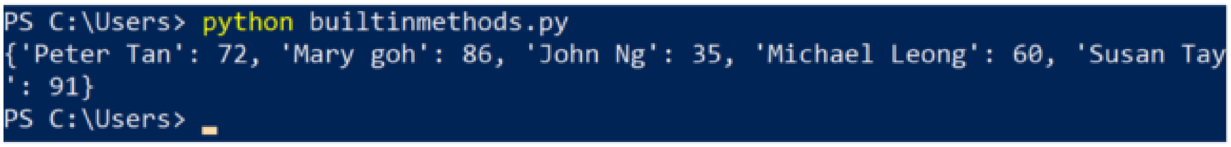
- https://www.w3schools.com/python/python_ref_tuple.asp
- https://www.w3schools.com/python/python_ref_list.asp
- https://www.w3schools.com/python/python_ref_dictionary.asp

- https://www.w3schools.com/python/python_ref_string.asp

Example (Cont'd): Now we also have each student's surname in our dictionary, and we want to sort the data according to it. In the first step, we would like to erase all the double spacing between the first and surname in the dictionary keys.



```python
class1 = {"Peter Tan": 72, "Mary goh": 86, "John Ng": 35}
class2 = {"Michael Leong": 60, "Susan  Tay": 91}
all_classes = class1 | class2
for i in list(all_classes.keys()):
    if i.find("  ") > 0:
        newkey = i.replace("  ", " ")
        all_classes[newkey] = all_classes[i]
        del all_classes[i]
print(all_classes)
```

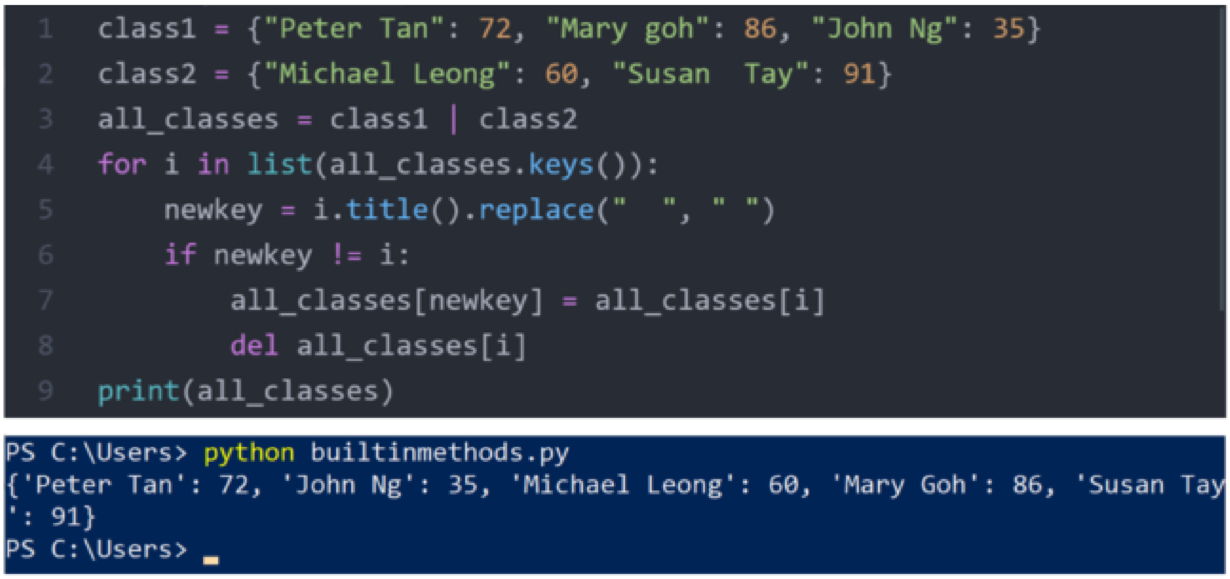


```
PS C:\Users> python builtinmethods.py
{'Peter Tan': 72, 'Mary goh': 86, 'John Ng': 35, 'Michael Leong': 60, 'Susan Tay
': 91}
PS C:\Users> _
```

Figure 2.42 Replacing Double Spacing by Single Spacing

In the first three lines, we again repeat the syntaxes from Chapter 1.3.2 to merge two dictionaries into a new one called `all_classes`. We implement a `for`-loop to iterate through all the dictionary keys and search for one with double spacing by the `.find()` method. If a key has double spacing, it will be replaced by single spacing. The `.replace(original_string, replacing_string)` method will take over this substitution process. Subsequently, Python assigns the original value to the new key and delete the old key from the dictionary.

In the next step, we would like to capitalise each word in the dictionary keys. It is important to mention that the `.capitalize()` method is not suitable for our task since it will only convert the first character in each key to upper case. Indeed, we need the `.title()` method to make sure that each word in a key starts with a capital letter after the entire process.



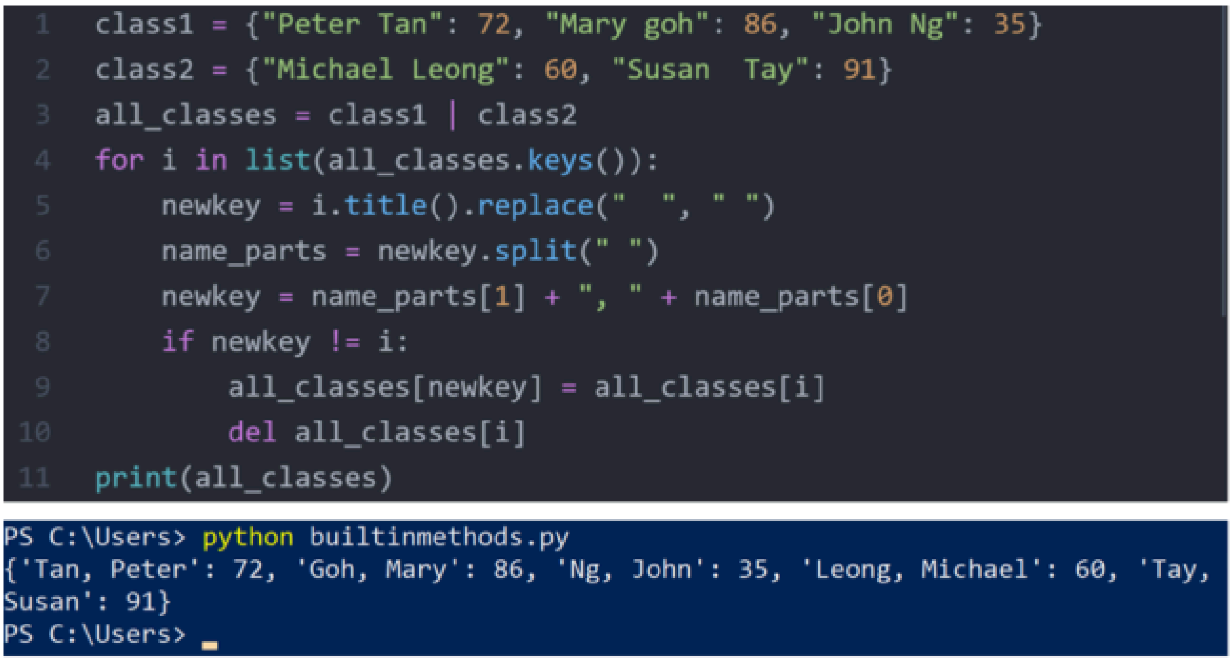
```python
class1 = {"Peter Tan": 72, "Mary goh": 86, "John Ng": 35}
class2 = {"Michael Leong": 60, "Susan  Tay": 91}
all_classes = class1 | class2
for i in list(all_classes.keys()):
    newkey = i.title().replace("  ", " ")
    if newkey != i:
        all_classes[newkey] = all_classes[i]
        del all_classes[i]
print(all_classes)
```

```
PS C:\Users> python builtinmethods.py
{'Peter Tan': 72, 'John Ng': 35, 'Michael Leong': 60, 'Mary Goh': 86, 'Susan Tay
': 91}
PS C:\Users> _
```

Figure 2.43 Capitalise Each Word in Strings

In the above program, we do not search and replace double spacing first and capitalise each word in the subsequent step. Instead, we initiate a new variable `newkey` which is the result of a chain execution of two methods in the same line. First, we convert the first letter of each word in the keys to upper case by the `.title()` method and then replace the double spacing by a single spacebar with the `.replace()` method. If the original key was correctly capitalised and spaced, the value stored in `newkey` must be the same as the original one. In this case, Python would do nothing and jump to the next key. Otherwise, Python will create a new key with the capitalisation and spacing correction and assign the value of the original key to it, and then delete the original key from the dictionary.

In the third step, we need to switch the format of the keys from "First name Last name" to "Last name, First name" for the subsequent sorting process (for simplicity, we assume that the students' names are all in the "First name Last name" format).



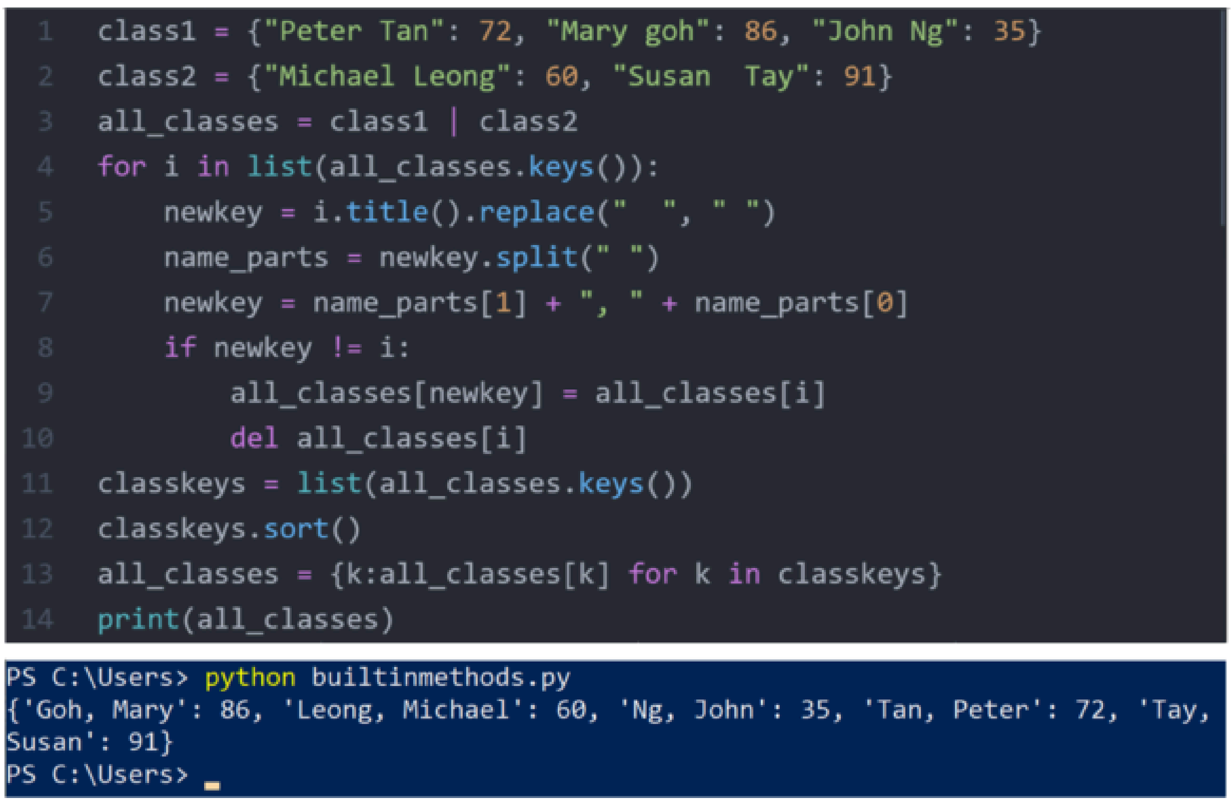
```python
class1 = {"Peter Tan": 72, "Mary goh": 86, "John Ng": 35}
class2 = {"Michael Leong": 60, "Susan  Tay": 91}
all_classes = class1 | class2
for i in list(all_classes.keys()):
    newkey = i.title().replace("  ", " ")
    name_parts = newkey.split(" ")
    newkey = name_parts[1] + ", " + name_parts[0]
    if newkey != i:
        all_classes[newkey] = all_classes[i]
        del all_classes[i]
print(all_classes)
```

```
PS C:\Users> python builtinmethods.py
{'Tan, Peter': 72, 'Goh, Mary': 86, 'Ng, John': 35, 'Leong, Michael': 60, 'Tay,
Susan': 91}
PS C:\Users> _
```

Figure 2.44 Switching Surname and First name in Strings

In Figure 2.44, we add two lines to format the dictionary keys according to our needs. First, we use the `.split()` method to separate each student's name into two parts: first name and last name, and the spacebar is used as the separation character of the string here. The result will be then stored as a list called `name_parts` with the structure ["First name", "Last name"]. In the next line, we swap the appearance order of first and last names by concatenating "Last name" and "First name" from the list and add a "`, `" between them to become a new string that is used as our final key. The replacement and delete process is identical to the previous one, and it should only be carried out if the original key and the new key are not identical.

Now, we can sort the dictionary by its keys in the ascending alphabetical order.



```
1   class1 = {"Peter Tan": 72, "Mary goh": 86, "John Ng": 35}
2   class2 = {"Michael Leong": 60, "Susan  Tay": 91}
3   all_classes = class1 | class2
4   for i in list(all_classes.keys()):
5       newkey = i.title().replace("  ", " ")
6       name_parts = newkey.split(" ")
7       newkey = name_parts[1] + ", " + name_parts[0]
8       if newkey != i:
9           all_classes[newkey] = all_classes[i]
10          del all_classes[i]
11  classkeys = list(all_classes.keys())
12  classkeys.sort()
13  all_classes = {k:all_classes[k] for k in classkeys}
14  print(all_classes)
```

```
PS C:\Users> python builtinmethods.py
{'Goh, Mary': 86, 'Leong, Michael': 60, 'Ng, John': 35, 'Tan, Peter': 72, 'Tay,
Susan': 91}
PS C:\Users> _
```

Figure 2.45 Sorting Dictionary by Its Keys

In line 11, the extracted dictionary keys are stored in a list called `classkeys` and this list is then sorted in the ascending order by the `.sort()` method in line 12. In line 13, we indicate that the contents of the dictionary `all_classes` will be “reassigned” by wrapping the expression on the right-hand side with a pair of braces. In the braces, we initiate a variable `k` that will store the sorted keys in `classkeys` once the `for`-loop starts to iterate. Behind the variable `k`, we use a colon to indicate that the expression following it is the value that belongs to the key `k` in `all_classes`. This syntax will then be followed by the `for`-loop mentioned before. It is important to use the same key variable, `k` in this case, in this `for`-loop.

It is noteworthy that no colon behind the `for`-statement will be needed if a `for`-loop is initiated this way. The instruction to be carried out in each iteration should be placed before the `for`-loop. By doing this, we can simplify our program by writing two commands in the same line instead of writing some lengthy syntaxes to initiate a `for`-loop to run through all the dictionary keys one by one in the traditional way.

Note that the sorting step can also be achieved with the same efficiency by the `sorted()` function. Since we only focus on the functionality and application of methods in this section, we try to use methods merely in our demonstration of the examples.

Read

Read the following website for more details and examples on Python methods for strings:

https://www.w3schools.com/python/python_ref_string.asp

Read the following website for more details and examples on Python methods for lists:

https://www.w3schools.com/python/python_ref_list.asp

Read the following website for more details and examples on Python methods for dictionaries:

https://www.w3schools.com/python/python_ref_dictionary.asp

Read the following website for more details and examples on Python methods for tuples:

https://www.w3schools.com/python/python_ref_tuple.asp

Chapter 3: User-defined Functions

Lesson Recording

User-defined Functions in Python

In Chapter 2, we are introduced to some built-in functions and methods that are already included in the Python programming environment. Functions and methods help us to carry out routine tasks which would require us to write very lengthy code to achieve the same functionality if we were to create the program by ourselves. Nevertheless, sometimes we can also write our own functions that suit our own needs.

A user-defined function (we will call it function in the following due to simplicity) can be viewed as a separate part of the code that will not be interpreted by Python until it is called from the main program. Usually, a function consists of four parts:

1. the function name
2. some arguments, i.e., values or parameters the function needs for its processing. This part is optional. If the function has no arguments, it will simply process all the instructions without any input from the main program.
3. the instructions of how and what to process within the function
4. an object (or a value) that should be returned to the main program at the end of the function. This is also optional. If no return object is specified, the main program will proceed without any output from the function.

In Python, the `def`-syntax indicates the definition of a function:

```
def function_name(argument1, argument2, …):
      instructions
      return object
```

Same as `for`-loops or `if`-conditions, the `def`-statement must end with a colon, and all the follow-up instructions and codes that belong to the function must be indented. Subsequently, the function can be called in the main program by integrating the `function_name` at an appropriate place.

```
… (Main program)
y = function_name(argument1 = object1, argument2 = object2,
 …)
print(y)
… (Continue with main program)
```

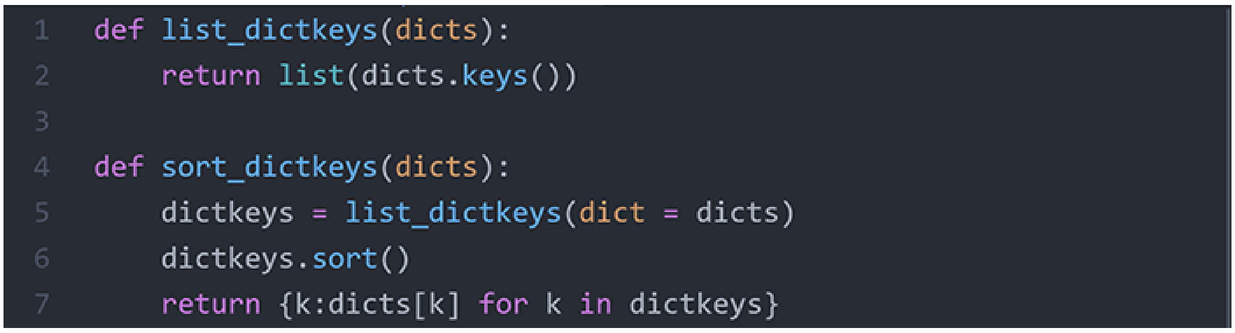
We use a function to carry out a certain process. The objects `object1` and `object2` are the corresponding input for `argument1` and `argument2` to the function. It is essential for Python that `object1` and `object2` are already defined somewhere in the previous part of the main program. The output object (or value) from the function will then be assigned to the variable `y`.

Caution! It is tempting for beginners to "outsource" some parts of the main program and make them separate functions for the sake of "program cleanliness". Though the code may look more structured at the first sight, the debugging process can be quite challenging if the whole program is jumping between the main program and the functions. The rules of thumb for using user-defined functions appropriately are:

1. if the same routine, probably with different arguments, appears more than once in the main program
2. if several functions should be combined into one which will then be used in the main program on multiple occasions
3. if a function can really increase the efficiency of the main program

In these cases, function can really simplify the program code and the debugging of it since there are less chances for syntax or logical errors.

Example (Cont'd): We repeat the same task from Chapter 2.2 by implementing two user-defined functions: `list_dictkeys(dicts)` and `sort_dictkeys(dicts)`.



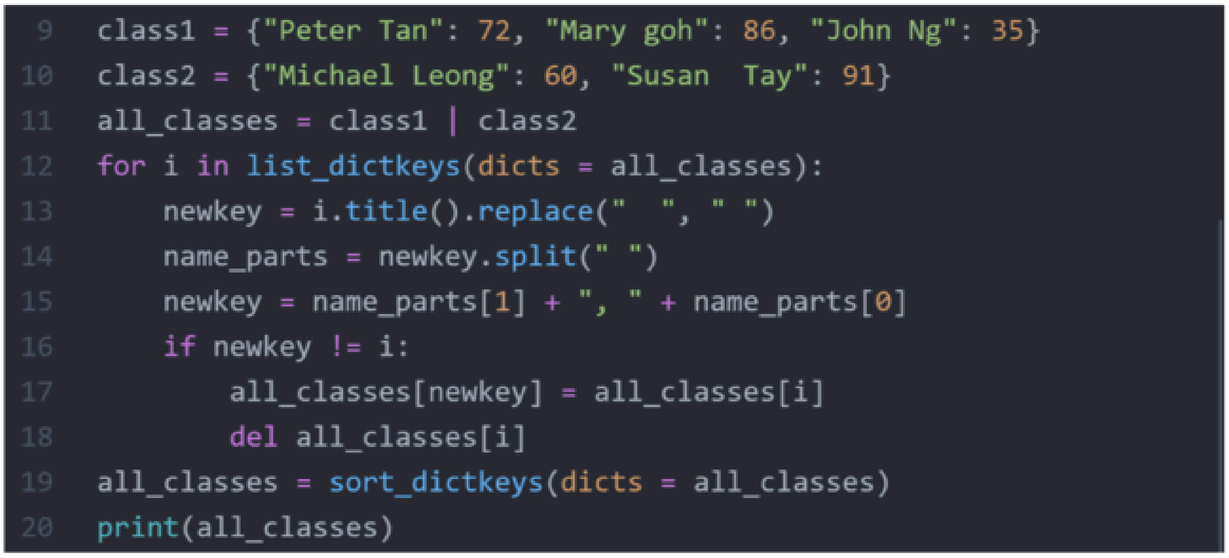
```
1  def list_dictkeys(dicts):
2      return list(dicts.keys())
3
4  def sort_dictkeys(dicts):
5      dictkeys = list_dictkeys(dict = dicts)
6      dictkeys.sort()
7      return {k:dicts[k] for k in dictkeys}
```

Figure 2.46 Defining Functions

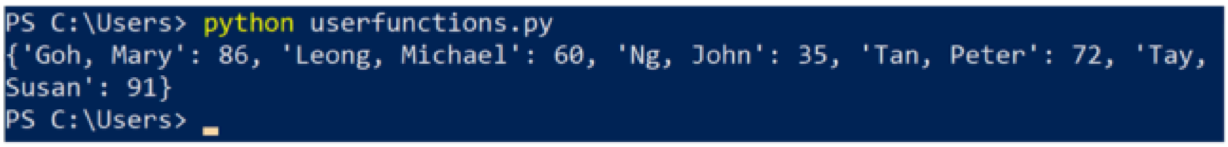
The first function `list_dictkeys(dicts)` is used to extract the keys from a dictionary and convert them to a list in the same step. It has an argument `dicts`, which is a dictionary variable to store the dictionary passed on by the main program to the function for the extraction and conversion process. This process is written in the `return` syntax directly since we target on returning the list as the output object of our function anyway. An additional step to save the list in an extra variable first and then to return the extra variable is therefore not necessary here.

The second function `sort_dictkeys(dicts)` sorts the dictionary keys directly. The code is basically identical to the lines 11, 12 and 13 in Figure 2.45. The only difference here is the dictionary object is not a specific dictionary from the main program, but a variable called `dicts` that is used as the argument in the definition of the function and is also only used within the function. The main program will then pass on a dictionary to `sort_dictkeys` and store the dictionary in `dicts` once it starts to process the instructions inside the function. It is noteworthy to mention that we can call another function within a function like in line 5, where we call `list_dictkeys(dicts)` to generate the list of dictionary keys for further process.

The main program does not become significantly shorter, but simpler in terms of the number of functions and methods involved.



```
 9  class1 = {"Peter Tan": 72, "Mary goh": 86, "John Ng": 35}
10  class2 = {"Michael Leong": 60, "Susan  Tay": 91}
11  all_classes = class1 | class2
12  for i in list_dictkeys(dicts = all_classes):
13      newkey = i.title().replace("  ", " ")
14      name_parts = newkey.split(" ")
15      newkey = name_parts[1] + ", " + name_parts[0]
16      if newkey != i:
17          all_classes[newkey] = all_classes[i]
18          del all_classes[i]
19  all_classes = sort_dictkeys(dicts = all_classes)
20  print(all_classes)
```



```
PS C:\Users> python userfunctions.py
{'Goh, Mary': 86, 'Leong, Michael': 60, 'Ng, John': 35, 'Tan, Peter': 72, 'Tay,
Susan': 91}
PS C:\Users>
```

Figure 2.47 Sorting Dictionary Using User-Defined Functions

In line 12, the program commands the list that should be run through in the `for`-loop is the output object returned from the `list_dictkeys(dicts)` function. In the bracket following the function name `list_dictkeys`, we specify the dictionary `all_classes` which should be stored in the variable `dicts` when passing on to `list_dictkeys(dicts)`. The same happened in line 19 where we use the `sort_dictkeys(dicts)` function to sort the dictionary `all_classes`. Note that the "`dicts =`" part is not required as long as the function only contains one argument, or when the objects to be passed on to the function are listed in the same sequence as the argument list written in the function code. It is therefore important to know either the sequence of the argument list of each function or to type the "`argument =`" part when calling a function.

Certainly, more from the main program can be "outsourced" to a function if those routines are called more frequently in the program. For instance, the process of formatting the students' name can be written as a user-defined function as well. Since defining such a function will not simplify our program here, we keep this part of the program as in Figure 2.45 due to the aforementioned guidelines of using user-defined functions appropriately.

Read

Read the following three exercises of the textbook for more details and examples on user-defined functions:

Exercise 18: Names, Variables, Code, Functions

Exercise 19: Functions and Variables

Exercise 21: Functions Can Return Something

Chapter 4: Modules, Packages and Libraries

Lesson Recording

Modules, Packages and Libraries in Python

4.1 Import a Standard Package

Beside build-in functions and methods, as well as user-defined functions, Python also provides packages which we can think of as a directory of Python scripts, the so-called modules. These modules specify new functions, methods, and object types for solving particular tasks. Packages are organised hierarchically; that means they may contain sub-packages, as well as regular modules themselves.

The packages of the standard library are already installed in the Python environment. A library is a collection of codes for us to perform specific tasks without writing our own code. But before we can use the modules in our program, we need to import the package or a specific module of the package first.

```
import package_name as package_alias
from package_name import module_name as module_alias
```

In the first syntax, we import the whole package into our program. The alias is a name that is used to refer to that particular package from thereon in our program. It is advantageous to use a package alias if it has a very long name. Note that the "`as package_alias`" part is optional in the import syntax. If the original package name is preferred, this part can be omitted.

The second syntax imports a particular module from a package. The alias here is the referral name of the module that we will use in our program, and not the package. Once again, the alias part is optional and can be omitted.

If the whole package is imported and we want to call a certain module from it, we will need to use the package name as the prefix and then indicate the module after a dot (`.`). The syntax should be something like:

```
y = package_name.module_name(argument1, argument2, …)
y = package_alias.module_name(argument1, argument2, …)
```

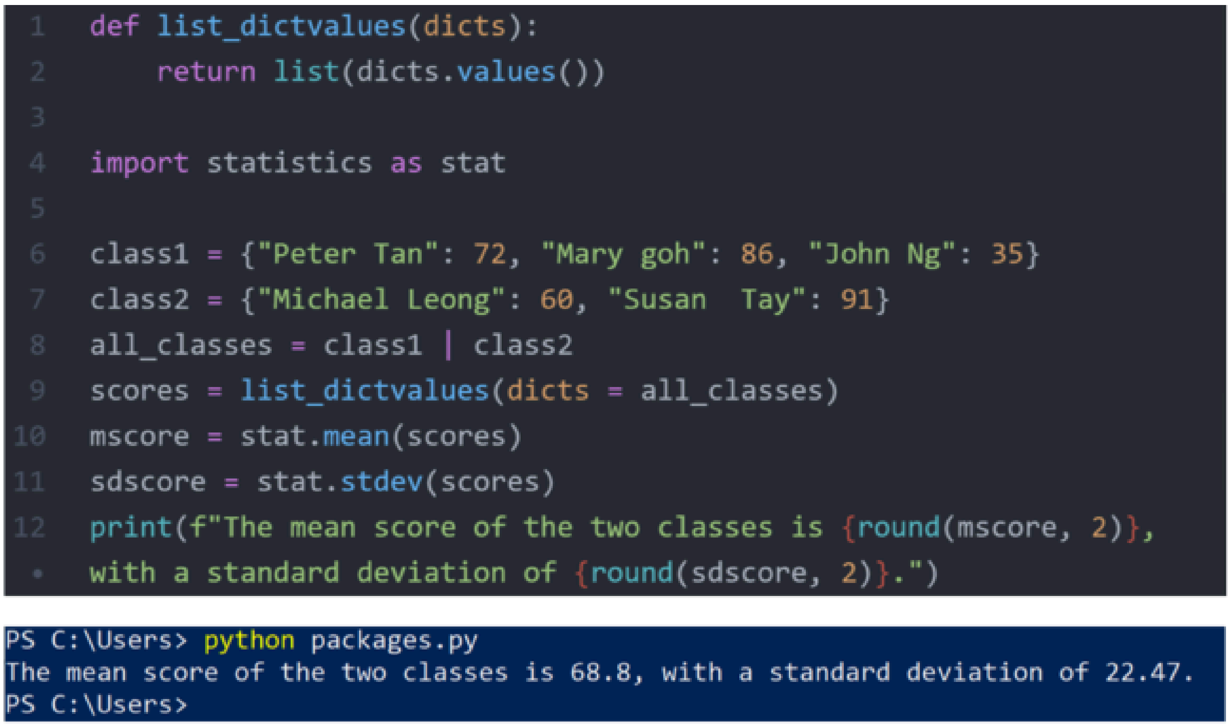
If we have used an alias for the package name in the import process, we will have to use the second syntax instead of the first one.

But if we have only imported a single module from a package, we could call it directly by its name without referring to the package:

```
y = module_name(argument1, argument2, …)
y = module_alias(argument1, argument2, …)
```

If we have only imported a single module from a package with an alias, we will have to use the alias instead of the original module name.

Example (Cont'd): If we need to calculate the mean and the standard deviation of the scores of the two classes, we can import the package `statistics` for this task.



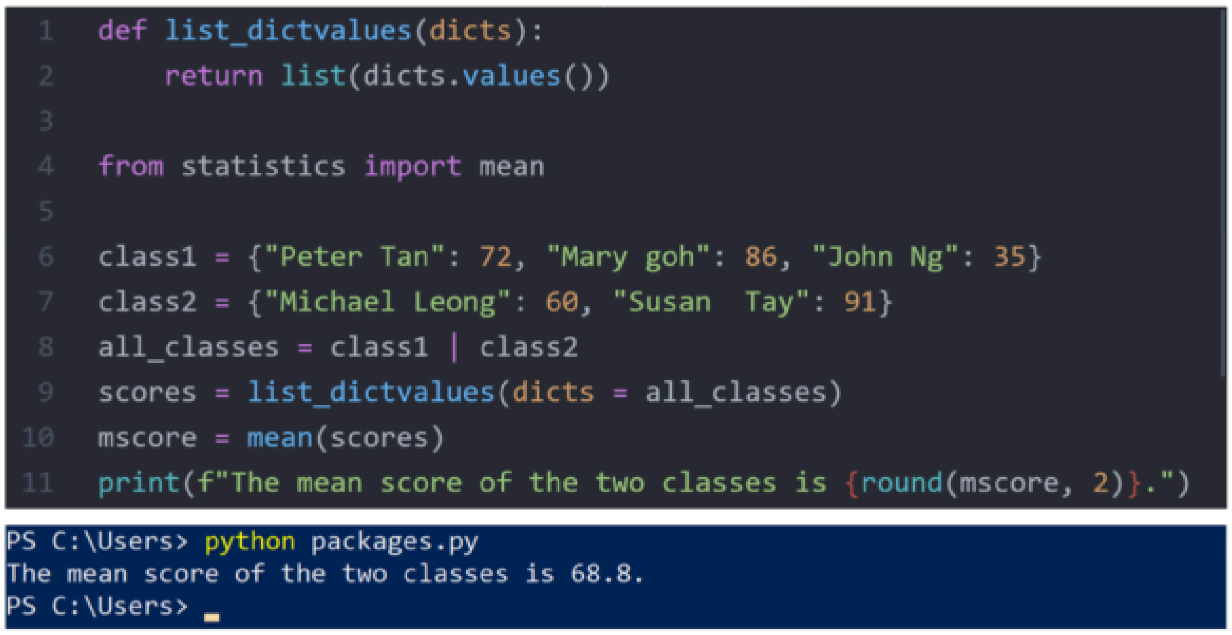
```python
1   def list_dictvalues(dicts):
2       return list(dicts.values())
3
4   import statistics as stat
5
6   class1 = {"Peter Tan": 72, "Mary goh": 86, "John Ng": 35}
7   class2 = {"Michael Leong": 60, "Susan  Tay": 91}
8   all_classes = class1 | class2
9   scores = list_dictvalues(dicts = all_classes)
10  mscore = stat.mean(scores)
11  sdscore = stat.stdev(scores)
12  print(f"The mean score of the two classes is {round(mscore, 2)},
 •  with a standard deviation of {round(sdscore, 2)}.")
```

```
PS C:\Users> python packages.py
The mean score of the two classes is 68.8, with a standard deviation of 22.47.
PS C:\Users>
```

Figure 2.48 Integrate Entire Package in a Program

We first create a function `list_dictvalues(dicts)` to extract the values from a dictionary and convert them to list in the same step. Subsequently, we import the package `statistics` and use `stat` as our alias to refer to it in our program. Since we have imported the whole package, we need to use the prefix `stat`. whenever we call a module from it, which then happens in line 10 and line 11. Line 10 contains the calculation of the mean by using the `stat.mean()` function, and in line 11, we instructed Python to compute the standard deviation by the `stat.stdev()` function.

If we just want to calculate the mean, we can then choose to import the `mean()` function from the `statistics` package instead.



```
1   def list_dictvalues(dicts):
2       return list(dicts.values())
3
4   from statistics import mean
5
6   class1 = {"Peter Tan": 72, "Mary goh": 86, "John Ng": 35}
7   class2 = {"Michael Leong": 60, "Susan  Tay": 91}
8   all_classes = class1 | class2
9   scores = list_dictvalues(dicts = all_classes)
10  mscore = mean(scores)
11  print(f"The mean score of the two classes is {round(mscore, 2)}.")
```

```
PS C:\Users> python packages.py
The mean score of the two classes is 68.8.
PS C:\Users> _
```

Figure 2.49 Integrate One Module from a Package

In line 4, we modify the syntax to the `from` ... `import` ... version. Since we feel that the module name `mean` does not need an alias, we keep the name as it is and call it in line 10 for the mean calculation.

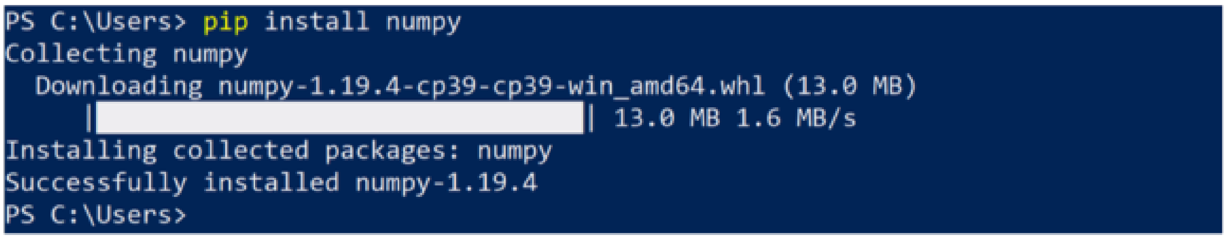
4.2 Managing Packages with pip/pip3

There are many Python packages available from the internet but not yet installed in the Python environment. To use those Python packages, we will first have to install them on our system. Then we can import them into our program same as the standard library. The simplest way to install such packages is to use PowerShell or Command Prompt as well (or similar terminal apps from other operating systems). Once we are prompted in the terminal window to give instructions to the operation system to carry out, type in one of the following commands and then press ENTER:

```
> pip install package_name
> pip3 install package_name
```

"pip" or "pip3" refers to the installer program that Python uses for installing external packages. Basically, "pip3" is a newer version of "pip". In most of the cases, we can use either one for our installation.

For instance, the package "numpy" will be needed in the next study units, and we would like to install it for our class preparation:

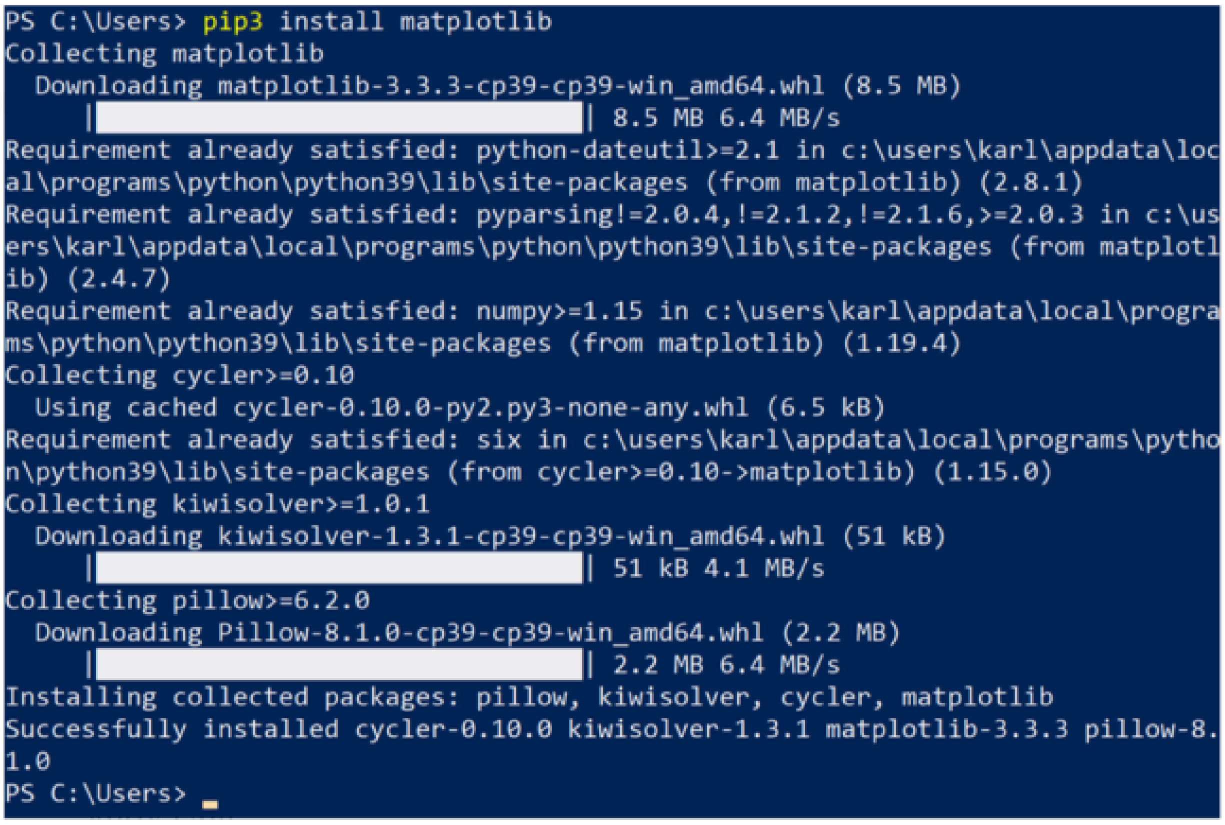


```
PS C:\Users> pip install numpy
Collecting numpy
  Downloading numpy-1.19.4-cp39-cp39-win_amd64.whl (13.0 MB)
     |████████████████████████████████| 13.0 MB 1.6 MB/s
Installing collected packages: numpy
Successfully installed numpy-1.19.4
PS C:\Users>
```

Figure 2.50 Installing Package "numpy" with pip

Once ENTER is pressed, Python will download the package installation file and install it subsequently. The message "Successfully installed xxx" will appear on the screen once the installation has been completed.

Another package that we will need in the next study units is "matplotlib". This time we install it with pip3.



```
PS C:\Users> pip3 install matplotlib
Collecting matplotlib
  Downloading matplotlib-3.3.3-cp39-cp39-win_amd64.whl (8.5 MB)
     |████████████████████████████████| 8.5 MB 6.4 MB/s
Requirement already satisfied: python-dateutil>=2.1 in c:\users\karl\appdata\loc
al\programs\python\python39\lib\site-packages (from matplotlib) (2.8.1)
Requirement already satisfied: pyparsing!=2.0.4,!=2.1.2,!=2.1.6,>=2.0.3 in c:\us
ers\karl\appdata\local\programs\python\python39\lib\site-packages (from matplotl
ib) (2.4.7)
Requirement already satisfied: numpy>=1.15 in c:\users\karl\appdata\local\progra
ms\python\python39\lib\site-packages (from matplotlib) (1.19.4)
Collecting cycler>=0.10
  Using cached cycler-0.10.0-py2.py3-none-any.whl (6.5 kB)
Requirement already satisfied: six in c:\users\karl\appdata\local\programs\pytho
n\python39\lib\site-packages (from cycler>=0.10->matplotlib) (1.15.0)
Collecting kiwisolver>=1.0.1
  Downloading kiwisolver-1.3.1-cp39-cp39-win_amd64.whl (51 kB)
     |████████████████████████████████| 51 kB 4.1 MB/s
Collecting pillow>=6.2.0
  Downloading Pillow-8.1.0-cp39-cp39-win_amd64.whl (2.2 MB)
     |████████████████████████████████| 2.2 MB 6.4 MB/s
Installing collected packages: pillow, kiwisolver, cycler, matplotlib
Successfully installed cycler-0.10.0 kiwisolver-1.3.1 matplotlib-3.3.3 pillow-8.
1.0
PS C:\Users> _
```

Figure 2.51 Installing Package "matplotlib" with pip3

In the same installation process, some other packages are downloaded and installed as well. This is because "matplotlib" needs some modules of these packages so that it can work.

If we wish to update/upgrade a package, the command that we need to type in our terminal app will be:

```
> pip install package_name --upgrade
> pip3 install package_name --upgrade
```

And if we want to uninstall a package, the command will become:

```
> pip uninstall package_name
> pip3 uninstall package_name
```

In Python, there are some packages that are used quite commonly for data analytics, and a couple of them will also be covered in this study guide.

Table 2.3 Some Common Packages for Data Analytics in Python

Package	**Description**	**In this Study Guide (Y/N)?**
matplotlib	Creates data visualisation	Y
numpy	Manages multi-dimensional arrays	Y
pandas	Handles two-dimensional data tables	Y
pendulum	Provides complex coding for dates and times	N
requests	Sends HTTP requests from Python code	N
scikit-learn	Provides tools of data analytics	Y
scipy	Carries out scientific and technical computations	N
sqlite3	Manages SQL database in Python	Y

Summary

In this unit, we have learned three types of built-in compound data structures of Python: tuples, lists and dictionaries. We discussed the major differences among these objects and the typical applications of them. Their creation and modification were also explained and demonstrated in detail. The most important issue here is the technique for subsetting and indexing the elements in these objects. We have also been introduced to functions, methods, and packages in Python. While some of them are built-in, i.e., they are already included in the Python environment, others can be user-defined or installed from external sources. Based on examples, we have been shown how built-in functions and methods, user-defined functions, as well as standard or external libraries can be applied to our Python programs.

Formative Assessment

1. What is the output of the following program?

```
a = 1, 4, 9, 16
a[2] = 3
print(a)
```

a. 1, 3, 9, 16

b. 1, 4, 3, 16

c. 1, 4, [3, 9], 16

d. Syntax Error

2. What is the output of the following program?

```
a = []
for i in range(0, 5):
    a = a + [i ** 2]
print(a)
```

a. `[0, 1, 2, 3, 4]`

b. `[0, 1, 4, 9, 16, 25]`

c. `[0, 1, 4, 9, 16]`

d. `[0, 1, 2, 3, 4, 5]`

3. Which of the following syntaxes will have “north” as output?

a.
```
d = {“north”: 2, “south”, -2}
dkeys = list(d.keys())
print(dkeys[0])
```

b.
```
d = {“north”: 2, “south”, -2}
dkeys = list(d.keys())
print(d[0])
```

c.
```
d = {“north”: 2, “south”, -2}
dkeys = list(d.items())
print(dkeys[0])
```

d.
```
d = {“north”: 2, “south”, -2}
dkeys = list(d.values())
print(dkeys[0])
```

4. Which of the following statements is correct regarding the properties of dictionaries?
   a. The values of a dictionary can only be integers, floats, and strings.
   b. The keys of a dictionary cannot be modified.
   c. The curly brackets around a dictionary can be omitted.
   d. The elements of a dictionary must be separated by semi-colons.

5. What is a Python function?
   a. A stand-alone Python program
   b. A dictionary with the module names of the standard library as keys
   c. An object specifically designed for machine learning operations
   d. A Python routine code that is reusable for a particular task

6. Which of the following methods does not apply to string variables?
   a. `.get()`
   b. `.lower()`

   c. `.replace()`
   d. `.strip()`

7. Which component is not optional in a user-defined function?
   a. Function name
   b. Arguments
   c. Loops
   d. Return value

8. What is not a good habit when implementing user-defined functions in a program?
   a. We should only implement user-defined functions whenever it is sensible to combine multiple functions into one.
   b. We should only implement user-defined functions when we need to carry out the same routine repeatedly in our program.
   c. We should only implement user-defined functions when we can reduce our main program to some syntaxes merely for calling the functions.
   d. We should only implement user-defined functions when we have recurrent tasks that create certain output objects needed for the further parts of the main program. And the creation of such objects requires some input arguments from the previous parts of the main program.

9. Which of the following statements is correct when using alias for importing package/module?
   a. We can use both the original name and alias to refer to the package/module in our program.
   b. An alias is optional and can be omitted if we are comfortable to work with the original package/module name.
   c. An alias must be shorter than the original package/module name.
   d. If only a single module from a package is imported, the alias refers to the package and not the module.

10. Which command is used to install a new Python package?

    a. `> pip3 setup package_name`

    b. `> pip3 install package_name --upgrade`

    c. `> pip3 install package_name`

    d. `> pip3 import package_name`

Solutions or Suggested Answers

Formative Assessment

1. What is the output of the following program?

```
a = 1, 4, 9, 16
a[2] = 3
print(a)
```

   a. 1, 3, 9, 16

   Incorrect. Since `a` is a tuple, it is immutable, and we cannot assign a new value to one of the elements in it.

   b. 1, 4, 3, 16

   Incorrect. Since `a` is a tuple, it is immutable, and we cannot assign a new value to one of the elements in it.

   c. 1, 4, [3, 9], 16

   Incorrect. Since `a` is a tuple, it is immutable, and we cannot assign a new value or change the type of the elements in it.

   d. Syntax Error

   Correct. Since `a` is a tuple, it is immutable. If we try to reassign a value to one of the elements in it, we will get an error message from Python.

2. What is the output of the following program?

```
a = []
for i in range(0, 5):
    a = a + [i ** 2]
```

```
print(a)
```

a. `[0, 1, 2, 3, 4]`

Incorrect. In each iteration, `i` square will be put into a list and then appended to a. And the values of `i` are 0, 1, 2, 3, 4.

b. `[0, 1, 4, 9, 16, 25]`

Incorrect. Though `i` square is taken here, the last value is out of range since the `for`-loop will stop running at `i = 4`.

c. `[0, 1, 4, 9, 16]`

Correct. The `for`-loop stops running at `i = 4` and the values in `a` are `i` square.

d. `[0, 1, 2, 3, 4, 5]`

Incorrect. The last value is out of range, and the values in `a` are `i` and not `i` square.

3. Which of the following syntaxes will have “north” as output?

a.
```
d = {“north”: 2, “south”, -2}
dkeys = list(d.keys())
print(dkeys[0])
```

Correct. The `.keys()` method is used to extract the keys of a dictionary. After converting the object to a list, we subset its item with the index 0, which is the first key in this case: “north”.

b.
```
d = {“north”: 2, “south”, -2}
dkeys = list(d.keys())
```

```
print(d[0])
```

Incorrect. In the `print()` function, it refers to the object `d` and not `dkeys`. And since `d` is dictionary, we can only access its element by the keys and not the indices.

c.

```
d = {“north”: 2, “south”, -2}
dkeys = list(d.items())
print(dkeys[0])
```

Incorrect. The `.items()` method is used to extract the keys and values of a dictionary and store each pair of them in a tuple. `dkeys[0]` will return `(“north”, 2)` as its result.

d.

```
d = {“north”: 2, “south”, -2}
dkeys = list(d.values())
print(dkeys[0])
```

Incorrect. The `.values()` method is used to extract the values of a dictionary. After converting the object to a list, we subset its item with the index 0, which is the first value 2 in this case, and not the key “north”.

4. Which of the following statements is correct regarding the properties of dictionaries?

   a. The values of a dictionary can only be integers, floats, and strings.

      Incorrect. The value of a dictionary can also be tuples, lists or other object types.

   b. The keys of a dictionary cannot be modified.

Correct. The keys of a dictionary cannot be modified. We can only modify a specific key indirectly by adding a new pair of key and value to the dictionary and delete the old pair from it.

c. The curly brackets around a dictionary can be omitted.

Incorrect. A Python dictionary must be wrapped by a pair of curly brackets.

d. The elements of a dictionary must be separated by semi-colons.

Incorrect. The elements of a dictionary must be separated by commas.

5. What is a Python function?

a. A stand-alone Python program

Incorrect. A function cannot be a stand-alone program. We must have a main program to call a function for a specific task.

b. A dictionary with the module names of the standard library as keys

Incorrect. A function is a routine program and not a dictionary.

c. An object specifically designed for machine learning operations

Incorrect. A function is not an object, it is a routine program.

d. A Python routine code that is reusable for a particular task

Correct. A function is a chunk of code that performs a particular task and can be called endlessly by the main program.

6. Which of the following methods does not apply to string variables?

a. `.get()`

Correct. It is a method for dictionary. It extracts the value of a particular dictionary key.

b. `.lower()`

Incorrect. It is a method for string variables. It converts a string to lower case.

c. `.replace()`

Incorrect. It is also a method for string variables. It replaces part of a string by another string.

d. `.strip()`

Incorrect. It is a method for string variables too. It removes all empty spaces at the beginning and at the end of a string.

7. Which component is not optional in a user-defined function?

a. Function name

Correct. A user-defined function must have a function name.

b. Arguments

Incorrect. Arguments are input from the main program to the function and they are optional. A function can still be carried out without arguments provided.

c. Loops

Incorrect. A function does not need loops for its functionality. Its instructions can contain loops, but it would still work without them.

d. Return value

Incorrect. A function can also be carried out even if it does not return any value to the main program.

8. What is not a good habit when implementing user-defined functions in a program?

a. We should only implement user-defined functions whenever it is sensible to combine multiple functions into one.

   Incorrect. It is a good habit indeed. If multiple functions are applied to a single object step-by-step, we can create a function and give a sensible name to it to simplify our main program.

b. We should only implement user-defined functions when we need to carry out the same routine repeatedly in our program.

   Incorrect. It is recommended to use user-defined functions to replace recurrent routines in the main program.

c. We should only implement user-defined functions when we can reduce our main program to some syntaxes merely for calling the functions.

   Correct. It is not a good habit to "outsource" chunks of code to various functions just for the sake of simplifying the main program. It usually does not simplify the whole program at all and makes the debugging more difficult since we must jump between the functions on checking the source of error.

d. We should only implement user-defined functions when we have recurrent tasks that create certain output objects needed for the further parts of the main program. And the creation of such objects requires some input arguments from the previous parts of the main program.

   Incorrect. It is a good habit to implement functions when they can really carry out recurrent tasks and return some objects/values to the main program based on some input arguments originated from the previous parts of the main program.

9. Which of the following statements is correct when using alias for importing package/ module?

a. We can use both the original name and alias to refer to the package / module in our program.

   Incorrect. Once we have imported a package / module with an alias, we must use the alias to refer to it in our program.

b. An alias is optional and can be omitted if we are comfortable to work with the original package / module name.

   Correct. The alias part is not compulsory. If we have a concise original package/module name, there is no reason to use an alias at all.

c. An alias must be shorter than the original package / module name.

   Incorrect. There is no guideline for the length of the alias. It can even be longer than the original package / module name.

d. If only a single module from a package is imported, the alias refers to the package and not the module.

   Incorrect. The alias refers to the module and not the package in the "`from` … `import` … `as` …" syntax.

10. Which command is used to install a new Python package?

    a. `> pip3 setup package_name`

       Incorrect. It should be "`install`" and not "`setup`" following "`pip3`".

    b. `> pip3 install package_name --upgrade`

       Incorrect. The "`--upgrade`"-option is only used for upgrading / updating existing packages.

    c. `> pip3 install package_name`

       Correct. "`install`" should be following "`pip3`" for installation.

d. `> pip3 import package_name`

   Incorrect. It should be “`install`” and not “`import`” following “`pip3`”.

References

Python.org. (n.d.). *Built-in exceptions*. Python Software Foundation. https://docs.python.org/3/library/stdtypes.html#typesmapping

Python.org. (n.d.). *Built-in functions*. Python Software Foundation. https://docs.python.org/3/library/functions.html

Python.org. (n.d.). *Built-in types*. Python Software Foundation. https://docs.python.org/3/library/stdtypes.html#sequence-types-list-tuple-range

Python.org. (n.d.). *Python dictionary methods*. Python Software Foundation. https://www.w3schools.com/python/python_ref_dictionary.asp

Python.org. (n.d.). *Python list methods*. Python Software Foundation. https://www.w3schools.com/python/python_ref_list.asp

Python.org. (n.d.). *Python string methods*. Python Software Foundation. https://www.w3schools.com/python/python_ref_string.asp

Python.org. (n.d.). *Python tuple methods*. Python Software Foundation. https://www.w3schools.com/python/python_ref_tuple.asp

Shaw, Z. A. (2017). *Learn python 3 the hard way*. Addison-Wesley Professional.

Study Unit 3

Arrays and Plots

Learning Outcomes

By the end of this unit, you should be able to:

1. Explain the operations on arrays
2. Analyse data using appropriate tools for data visualisation

Overview

This study unit introduces two Python packages: NumPy and matplotlib. NumPy is the fundamental package for efficient scientific computing with Python. We will learn how to create NumPy arrays and how to use indexing and Boolean masks for subsetting NumPy arrays. We will also learn the NumPy functions to generate statistics on the data stored in an array. Furthermore, we will also learn how to use the "matplotlib.pyplot" sub-package for data visualisation purpose. In particular, the functionalities available for plotting and customising basic charts of data analytics will also be a main focus of this study unit.

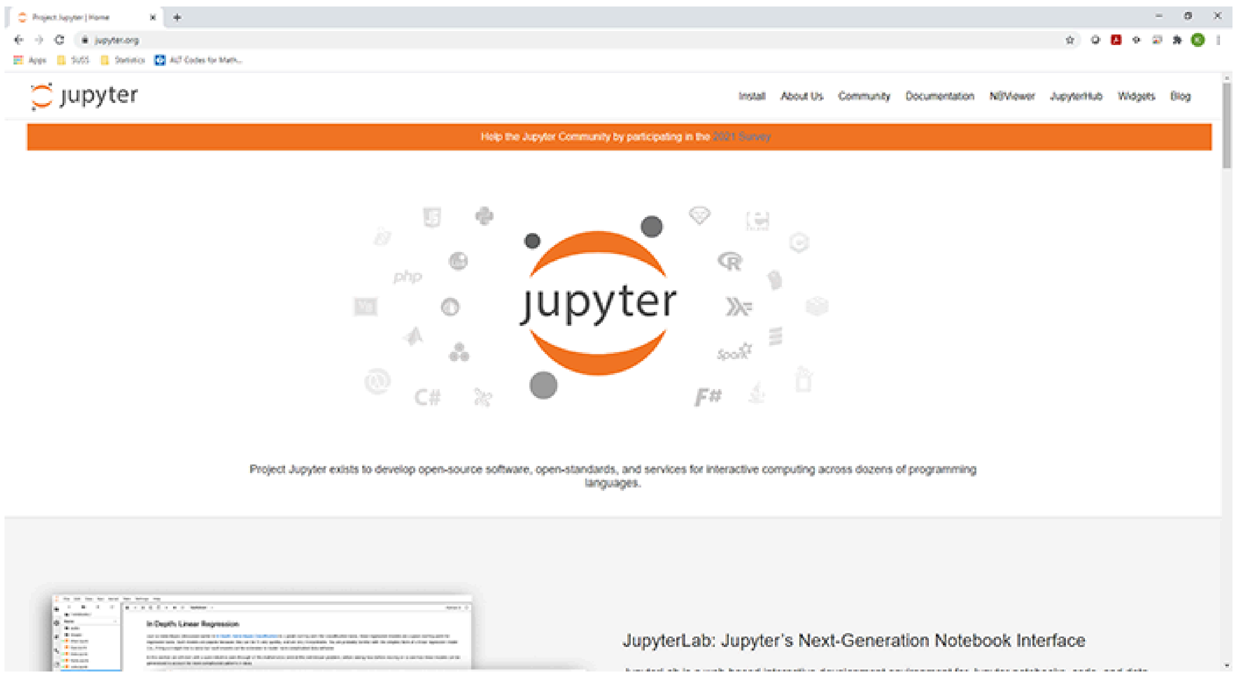
Chapter 1: Introduction to JupyterLab

Lesson Recording

Introduction to JupyterLab

In the previous study units, we write our programs in Atom first and run them in terminal apps such as PowerShell or Command Prompt. Starting from this study unit, our focus will shift from general programming to Python programming for data analytics. For this purpose, we will work with another Python programming environment called the JupyterLab.

While Atom is more a Python code editor in the traditional sense, JupyterLab is an open-source web application specialised in data analytics using Python. It is the newest Python programming interface developed by Project Jupyter. We can use it to create code for cleaning and transforming data, running numerical simulation, performing statistical modelling, data visualisation and machine learning (https://jupyter.org/).


Project Jupyter | Home
jupyter.org
jupyter
Install About Us Community Documentation NBViewer JupyterHub Widgets Blog
Help the Jupyter Community by participating in the 2021 Survey
jupyter
JupyterLab: Jupyter's Next-Generation Notebook Interface

Project Jupyter exists to develop open-source software, open-standards, and services for interactive computing across dozens of programming languages.

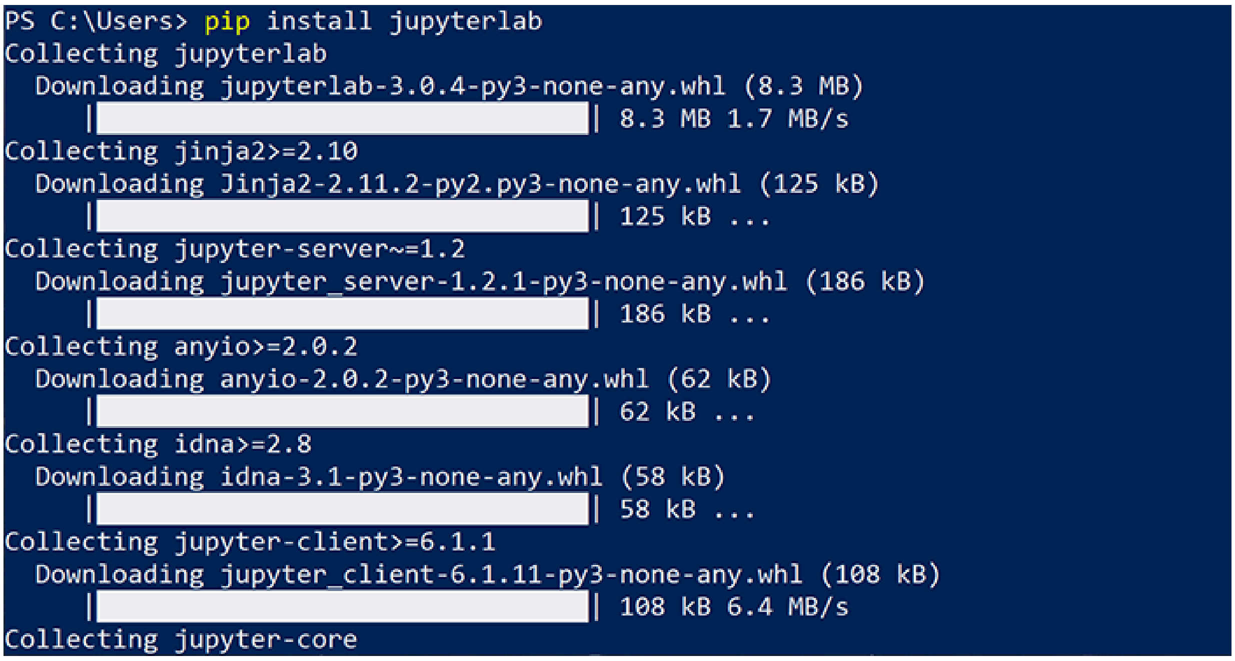
Figure 3.1 The Official Website for Jupyter

1.1 Installing JupyterLab

Before we can start working with JupyterLab, we need to install it on our computer so that it can be integrated in the Python environment. We need to type in the following command in our terminal app for its installation:

```
pip install jupyterlab
```

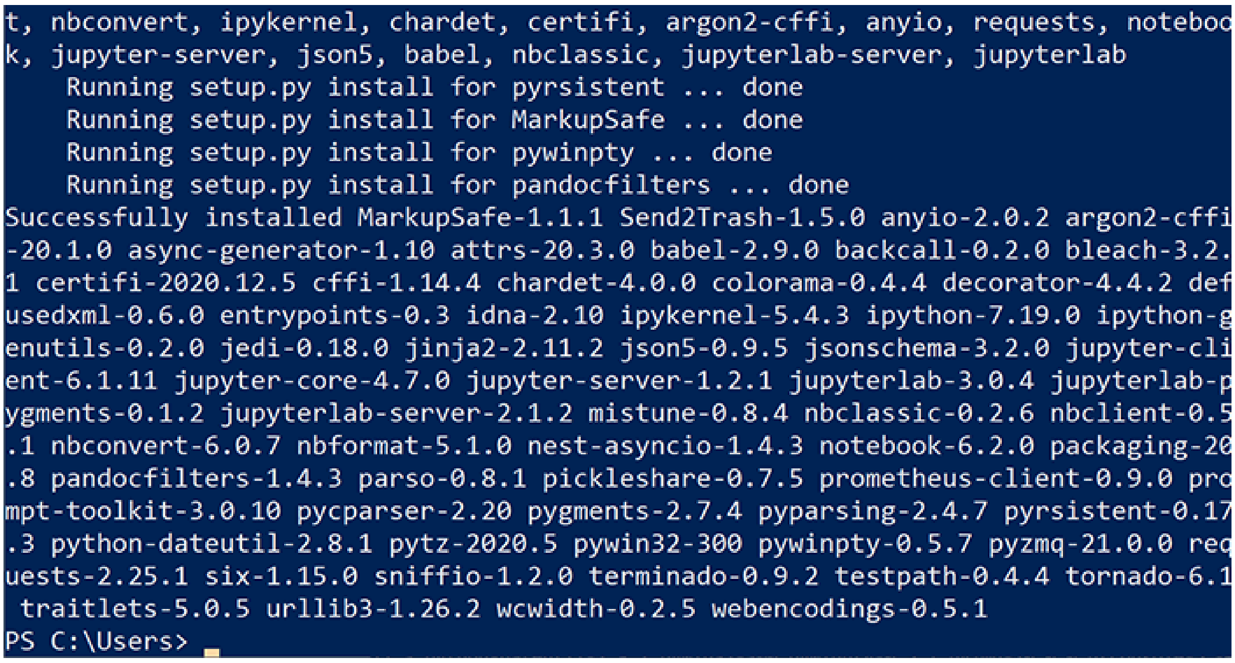
The `pip install` command refers to the same Python installer program introduced for package installation in Chapter 4.2 of Study Unit 2. Instead of a package, JupyterLab is the object to be installed here.



```
PS C:\Users> pip install jupyterlab
Collecting jupyterlab
  Downloading jupyterlab-3.0.4-py3-none-any.whl (8.3 MB)
     |████████████████████████████████| 8.3 MB 1.7 MB/s
Collecting jinja2>=2.10
  Downloading Jinja2-2.11.2-py2.py3-none-any.whl (125 kB)
     |████████████████████████████████| 125 kB ...
Collecting jupyter-server~=1.2
  Downloading jupyter_server-1.2.1-py3-none-any.whl (186 kB)
     |████████████████████████████████| 186 kB ...
Collecting anyio>=2.0.2
  Downloading anyio-2.0.2-py3-none-any.whl (62 kB)
     |████████████████████████████████| 62 kB ...
Collecting idna>=2.8
  Downloading idna-3.1-py3-none-any.whl (58 kB)
     |████████████████████████████████| 58 kB ...
Collecting jupyter-client>=6.1.1
  Downloading jupyter_client-6.1.11-py3-none-any.whl (108 kB)
     |████████████████████████████████| 108 kB 6.4 MB/s
Collecting jupyter-core
```

Figure 3.2 Start Installing JupyterLab

The installation could take quite a while since there are many packages that JupyterLab requires for its functionalities, and they will therefore be installed together.



```
t, nbconvert, ipykernel, chardet, certifi, argon2-cffi, anyio, requests, noteboo
k, jupyter-server, json5, babel, nbclassic, jupyterlab-server, jupyterlab
    Running setup.py install for pyrsistent ... done
    Running setup.py install for MarkupSafe ... done
    Running setup.py install for pywinpty ... done
    Running setup.py install for pandocfilters ... done
Successfully installed MarkupSafe-1.1.1 Send2Trash-1.5.0 anyio-2.0.2 argon2-cffi
-20.1.0 async-generator-1.10 attrs-20.3.0 babel-2.9.0 backcall-0.2.0 bleach-3.2.
1 certifi-2020.12.5 cffi-1.14.4 chardet-4.0.0 colorama-0.4.4 decorator-4.4.2 def
usedxml-0.6.0 entrypoints-0.3 idna-2.10 ipykernel-5.4.3 ipython-7.19.0 ipython-g
enutils-0.2.0 jedi-0.18.0 jinja2-2.11.2 json5-0.9.5 jsonschema-3.2.0 jupyter-cli
ent-6.1.11 jupyter-core-4.7.0 jupyter-server-1.2.1 jupyterlab-3.0.4 jupyterlab-p
ygments-0.1.2 jupyterlab-server-2.1.2 mistune-0.8.4 nbclassic-0.2.6 nbclient-0.5
.1 nbconvert-6.0.7 nbformat-5.1.0 nest-asyncio-1.4.3 notebook-6.2.0 packaging-20
.8 pandocfilters-1.4.3 parso-0.8.1 pickleshare-0.7.5 prometheus-client-0.9.0 pro
mpt-toolkit-3.0.10 pycparser-2.20 pygments-2.7.4 pyparsing-2.4.7 pyrsistent-0.17
.3 python-dateutil-2.8.1 pytz-2020.5 pywin32-300 pywinpty-0.5.7 pyzmq-21.0.0 req
uests-2.25.1 six-1.15.0 sniffio-1.2.0 terminado-0.9.2 testpath-0.4.4 tornado-6.1
 traitlets-5.0.5 urllib3-1.26.2 wcwidth-0.2.5 webencodings-0.5.1
PS C:\Users> _
```

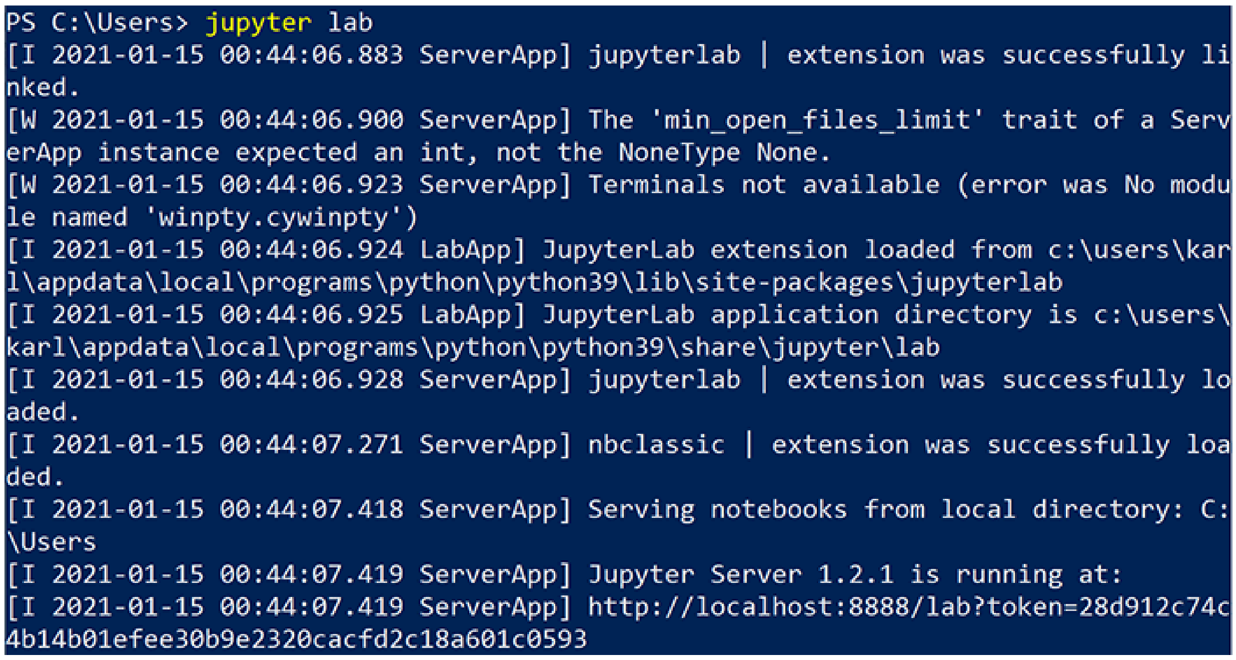
Figure 3.3 Installation of Jupyter Completed

The message “Successfully installed …” will appear on the terminal app once the installation of JupyterLab has completed.

1.2 Starting JupyterLab

To launch JupyterLab, we need to start the terminal app again, change to the folder where you have saved all your Python scripts, and then type in the following command:

```
jupyter lab
```



```
PS C:\Users> jupyter lab
[I 2021-01-15 00:44:06.883 ServerApp] jupyterlab | extension was successfully li
nked.
[W 2021-01-15 00:44:06.900 ServerApp] The 'min_open_files_limit' trait of a Serv
erApp instance expected an int, not the NoneType None.
[W 2021-01-15 00:44:06.923 ServerApp] Terminals not available (error was No modu
le named 'winpty.cywinpty')
[I 2021-01-15 00:44:06.924 LabApp] JupyterLab extension loaded from c:\users\kar
l\appdata\local\programs\python\python39\lib\site-packages\jupyterlab
[I 2021-01-15 00:44:06.925 LabApp] JupyterLab application directory is c:\users\
karl\appdata\local\programs\python\python39\share\jupyter\lab
[I 2021-01-15 00:44:06.928 ServerApp] jupyterlab | extension was successfully lo
aded.
[I 2021-01-15 00:44:07.271 ServerApp] nbclassic | extension was successfully loa
ded.
[I 2021-01-15 00:44:07.418 ServerApp] Serving notebooks from local directory: C:
\Users
[I 2021-01-15 00:44:07.419 ServerApp] Jupyter Server 1.2.1 is running at:
[I 2021-01-15 00:44:07.419 ServerApp] http://localhost:8888/lab?token=28d912c74c
4b14b01efee30b9e2320cacfd2c18a601c0593
```

Figure 3.4 Starting the JupyterLab

The messages appearing in the terminal app are no longer relevant to our work, unless we receive an error message from Python for loading JupyterLab. Under normal circumstances, the JupyterLab environment will be launched automatically in a new window or a new tab of the standard internet browser. If it does not start by itself, we can start the internet browser manually and type in the following URL in the address bar:

```
localhost:8888/lab
```

The start-up page of JupyterLab will then be loaded.

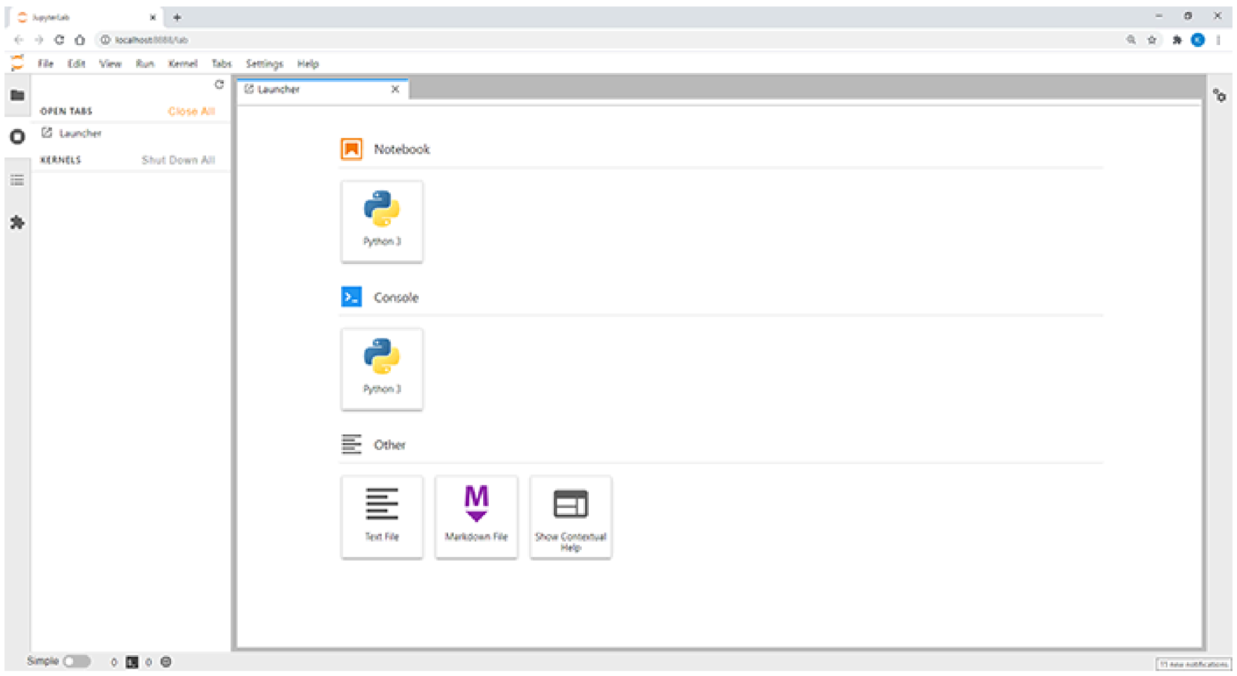


Figure 3.5 Start-Up Page of JupyterLab

To start a new Python script, you can press on the "Python 3" button in the "Notebook" rubric. And a new tab will appear in JupyterLab.

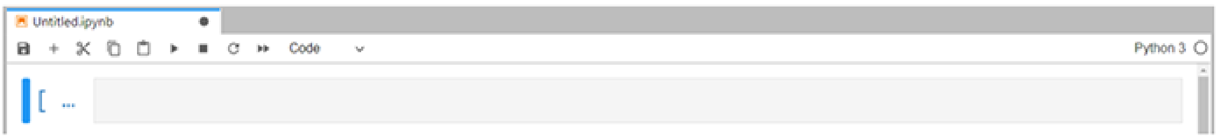
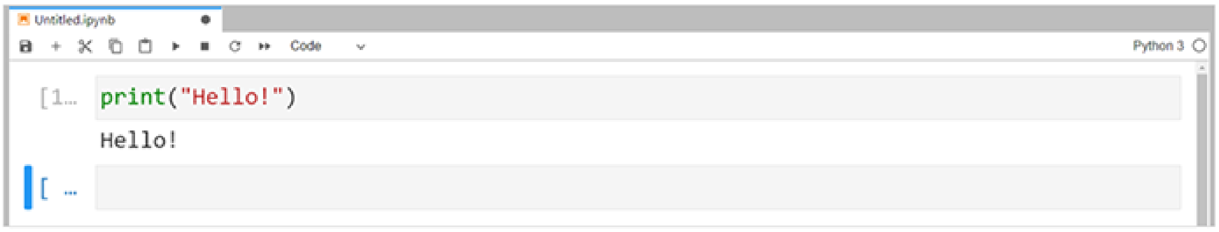


Figure 3.6 Blank Python Script

When this page appears, we can start writing our program in the field with a thick blue bar on the left end.

1.3 Working with JupyterLab

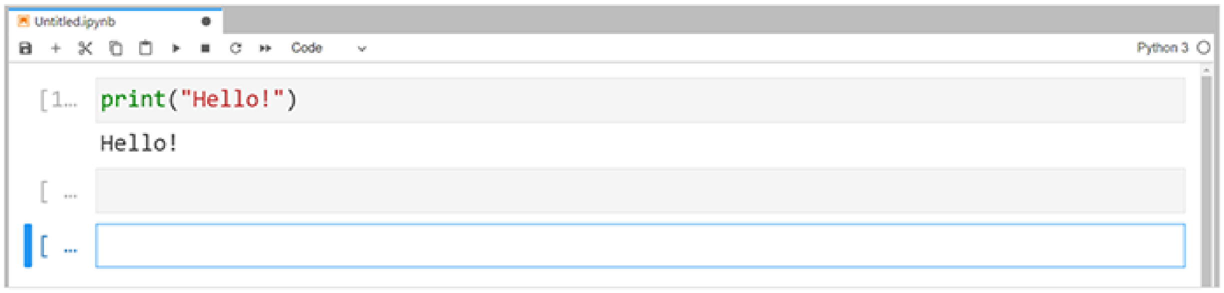
Each Python program written in a JupyterLab cell can be executed by clicking ▶ or pressing the key combination CTRL + ENTER. The output of the program script will then be printed below the input box as illustrated in Figure 3.7.



```
[1… print("Hello!")
    Hello!
[ …
```

Figure 3.7 Running a Python Script in JupyterLab

After running the first script, JupyterLab will usually add a new cell for us to start another task. Nevertheless, we can also add it manually by pressing + .



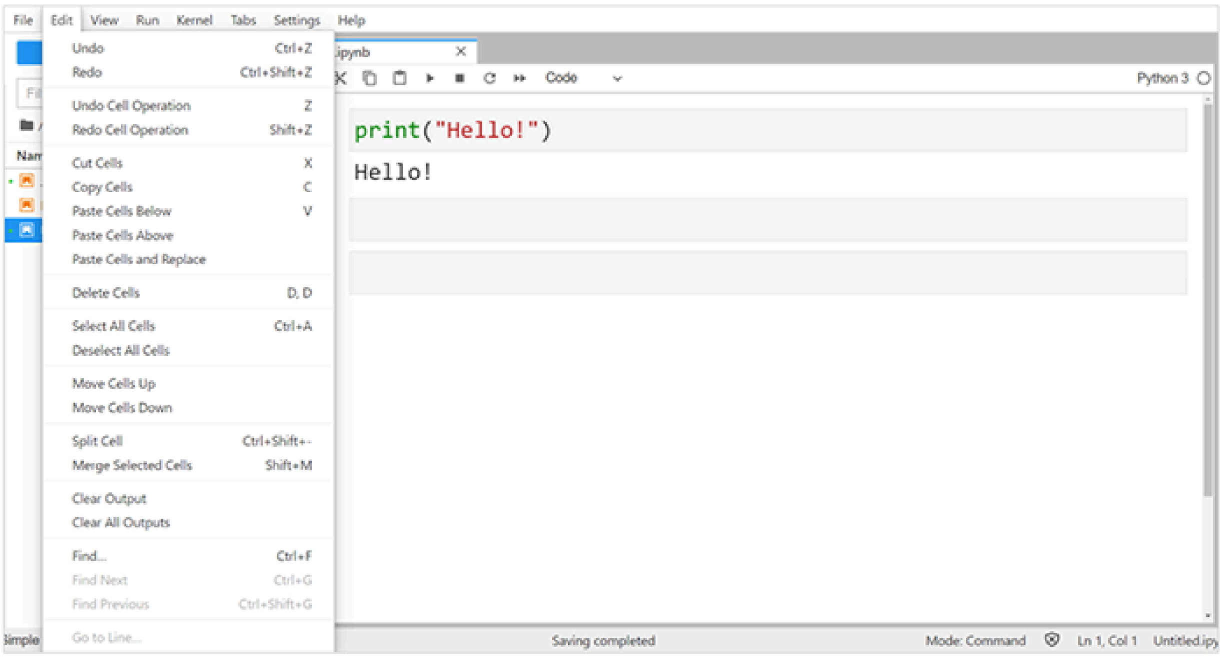
```
[1… print("Hello!")
    Hello!
[ …
[ …
```

Figure 3.8 Inserting a New Cell in JupyterLab

Once a new cell has been inserted, we can write another set of script in it. We can also choose to go back to the previous cell and modify the code written there.

Note that in JupyterLab, Python only executes the code written in *one* cell. In other words, we can return to the "upper" cells and re-run the code there when it is necessary. If we want to execute the programs in all cells, we can go to the "Kernel" menu and select "Restart Kernel & Run All Cells…". In this case, we have to pay attention to the sequence of the cells since the logical flow among them will become relevant.

In the "Edit" menu, there are many functions that JupyterLab provides to restructure our Python scripts. For example, we can switch the order of the cells by moving them up and down. We can also cut, copy, paste, and delete them. JupyterLab enables us to merge multiple cells into one or split a single cell into two or more cells as well.

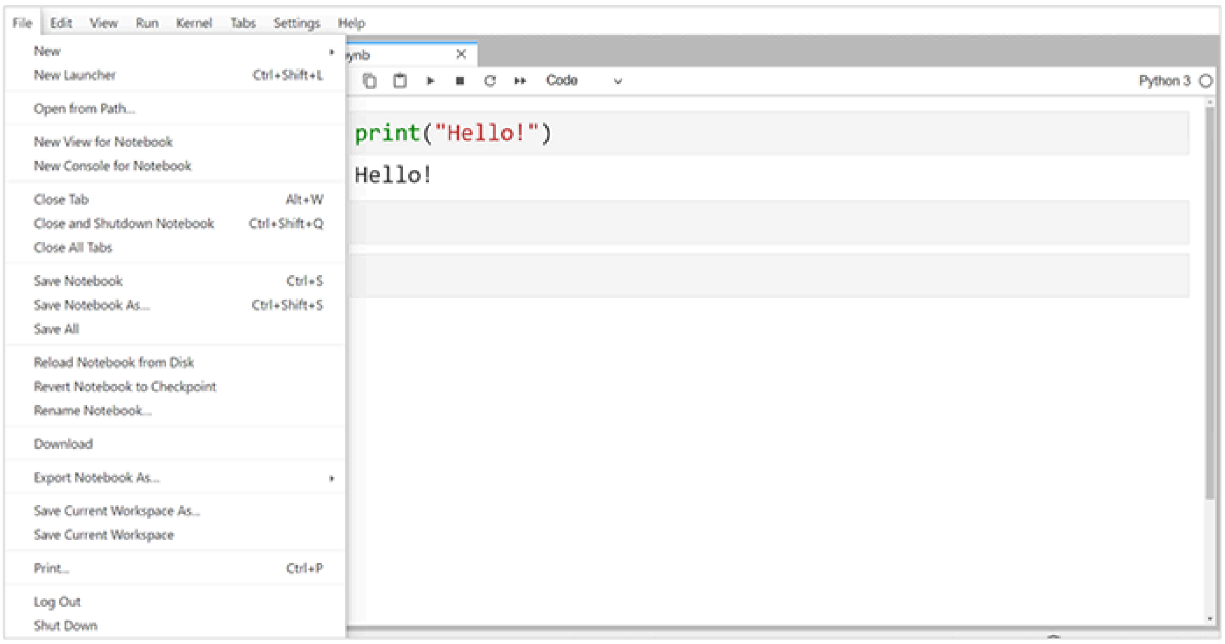


```
print("Hello!")
```

```
Hello!
```

Figure 3.9 Functions Included in the Edit Menu of JupyterLab

To save a Python program in JupyterLab, we can either press or choose "Save Notebook As…" in the "File" menu. The file will then be saved with the ".ipynb" extension in the folder where JupyterLab was started in the terminal app.



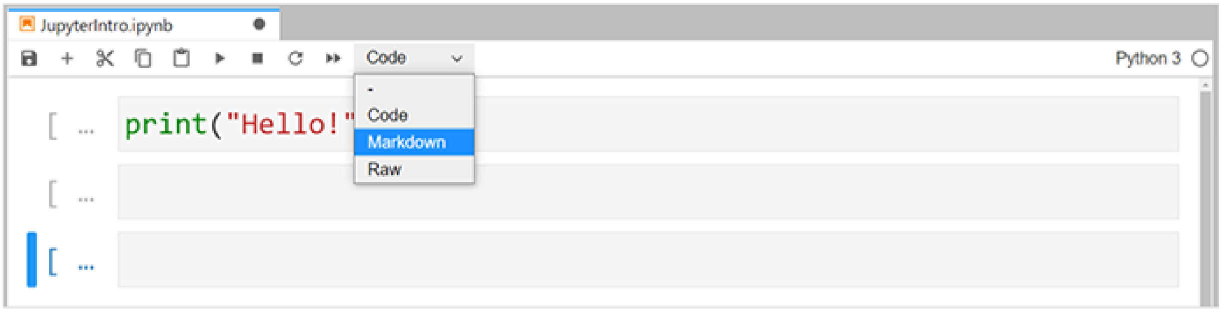
```
print("Hello!")
```

```
Hello!
```

Figure 3.10 Saving Python Program in JupyterLab

1.4 Markdown

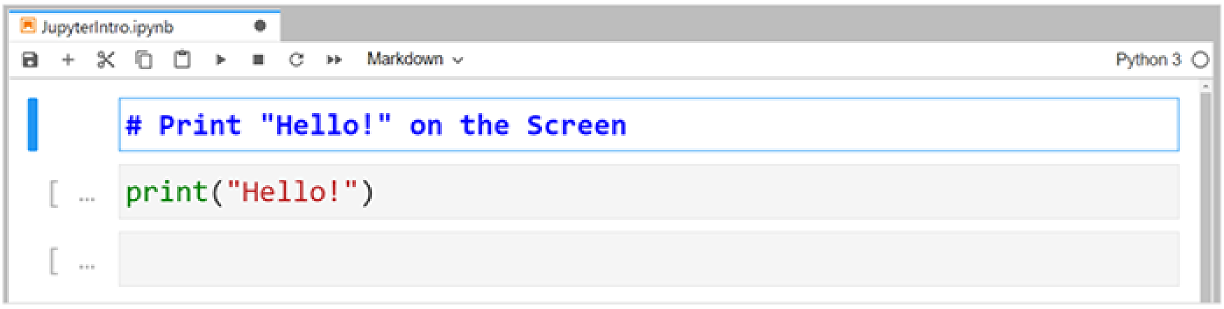
Another advantage of using JupyterLab is that we can use it as an advanced text editor. Besides Python programs, we can also embed elaborative texts to the program or write HTML codes to design a website with it. For this purpose, we need to switch the cell type from "Code" to "Markdown".



```
print("Hello!"
```

Figure 3.11 Changing the Cell Type in JupyterLab

For instance, we can write our Python comments introduced in Study Unit 1 in a markdown cell for explanatory purpose. In this case, we put a hash (#) at the beginning of the cell after converting it from a code cell to a markdown cell.

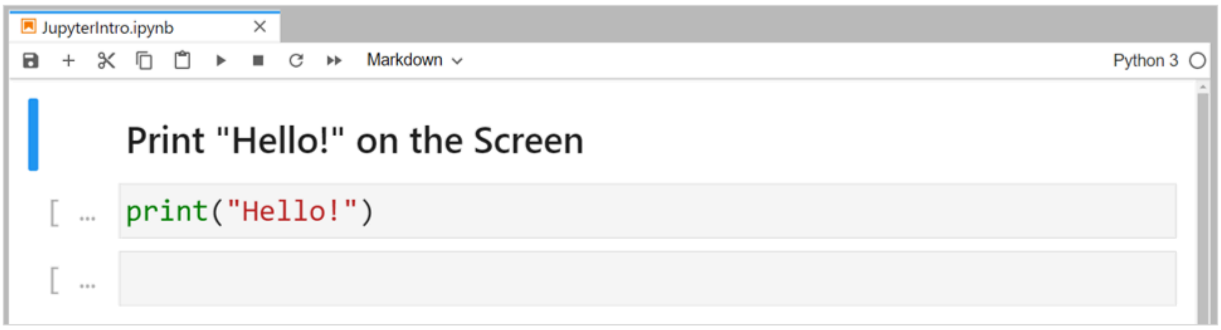


```
# Print "Hello!" on the Screen
```

```
print("Hello!")
```

Figure 3.12 Editing a Markdown Cell in JupyterLab

After editing the comment, we can press CTRL + ENTER to finalise the cell. The comment will be formatted as a header with bold and large font.

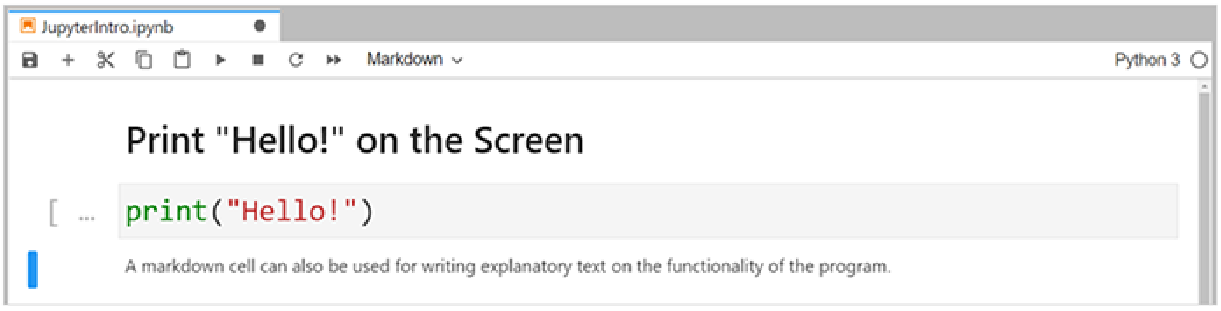


Print "Hello!" on the Screen

```
print("Hello!")
```

Figure 3.13 A Finalised Markdown Cell with a Comment in JupyterLab

If we do not start the markdown cell with the hash (#), JupyterLab will interpret the content as ordinary text and print it in the standard text format to the Python script file after finalising the cell.



Print "Hello!" on the Screen

```
print("Hello!")
```

A markdown cell can also be used for writing explanatory text on the functionality of the program.

Figure 3.14 A Finalised Markdown Cell with Ordinary Text in JupyterLab

Read

Read the following website for detailed explanation of JupyterLab including all the functionalities, configurations, and examples:

https://jupyterlab.readthedocs.io/en/stable/

Chapter 2: Array Management with NumPy

Lesson Recording

Array Management with NumPy

In Study Unit 2, we have been introduced to various types of compound data such as tuples, lists and dictionaries. We have also discussed in detail on modifying a list or storing different types of elements in a list. Despite being able to group lists or tuples in a superordinate list, these types of compound data are basically still one-dimensional. Recall in one part of our exam score example in the previous study units, the list of data to be analysed can consist of either sub-lists with individual student scores from different subjects or the scores of all students in one subject in each one of them. As a result, the data in this example are in fact two-dimensional: the student dimension and the subject dimension.

Hence, lists and dictionaries are no longer sufficient to store multidimensional data for analysis, and arrays should be used instead. Nevertheless, we can also replace lists and dictionaries by arrays when it comes to one-dimensional data. Note that the shape of a Python array must be rectangular, that is, the number of values in each row and each column must be identical, and all values in it must be entirely of the same type, typically numeric values or strings. Therefore, an array is not equivalent to a dataset in the conventional sense since missing values and mixed data types are common features of usual datasets. In Python, we can work with arrays using the “numpy” package.

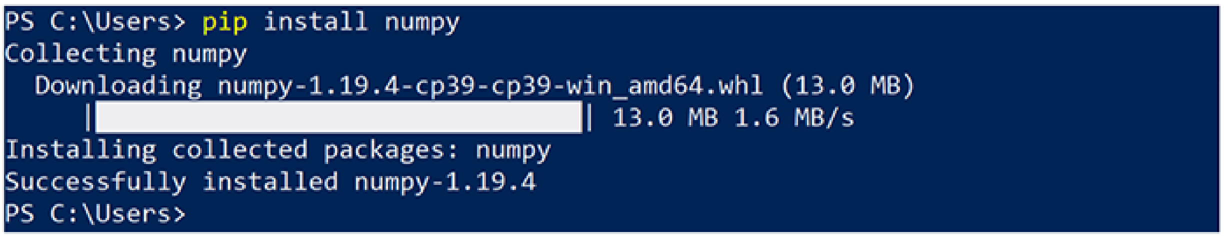
2.1 Creating NumPy Arrays

In order to work with NumPy arrays, we need to install the “numpy” package in Python first. We have already explained how to install packages in Python using pip or pip3 in

Chapter 4.2 of Study Unit 2. To summarise the procedure quickly, we have to launch the terminal app and type in the following command:

```
> pip install numpy
```

The installation will follow as shown in Figure 3.15.



```
PS C:\Users> pip install numpy
Collecting numpy
  Downloading numpy-1.19.4-cp39-cp39-win_amd64.whl (13.0 MB)
     |████████████████████████████████| 13.0 MB 1.6 MB/s
Installing collected packages: numpy
Successfully installed numpy-1.19.4
PS C:\Users>
```

Figure 3.15 Installing Package "numpy"

Once the installation is completed, we can launch JupyterLab and start working with NumPy arrays after the "numpy" package has been imported into our program. The corresponding syntax in Python is:

```
import numpy as np
```

Recall that we can import a package with an alias, which will then be used as our reference to the package in the further part of our program. And for "numpy", the most common alias used in the literature or online references is "np".

To create an array, we can use the `array()` function:

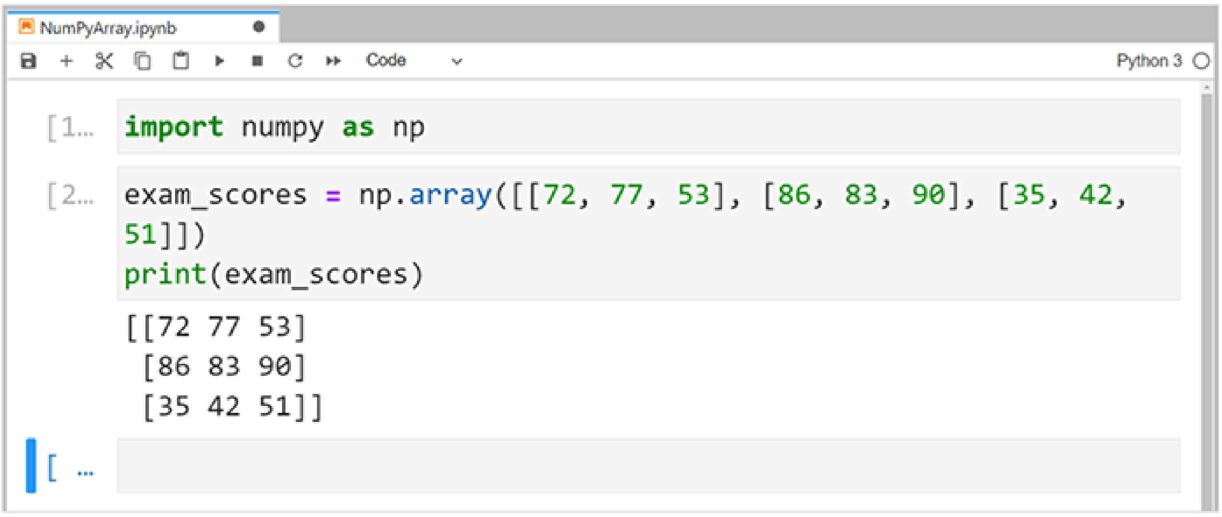
```
array_name = np.array([[list1_data1, list1_data2, …],
                       [list2_data1, list2_data2, …], …])
```

In the above syntax, the `array()` function was attached to the `np.` prefix, which is required if NumPy is imported using the `import`-statement. The prefix should be omitted if we use the `from … import …` statement to import NumPy instead.

The data assigned to `array_name` in the `array()` function are stored in various regular Python lists originally. Each list corresponds to a row of the array, and the total number of rows is therefore equal to the number of lists included in the `array()` function. Additionally, the number of elements in each list must be identical. If the lists have different lengths, an error message will appear. Note that it is compulsory to wrap all the lists, separated by commas, in a pair of outer square brackets again before putting them into the `array()` function.

In NumPy, an n-dimensional array is also called an "ndarray". Each direction of an array is called an axis. For instance, the rows or the columns are the two axes of a two-dimensional array, just like the coordinate system.

Example (Student score, cont'd): Suppose we have three lists and each of them contains one student's exam scores of three different courses. The three lists are `[72, 73, 53]`, `[86, 83, 90]` and `[35, 42, 51]`. And we would like to store all these scores in a two-dimensional array.



```
[1… import numpy as np

[2… exam_scores = np.array([[72, 77, 53], [86, 83, 90], [35, 42,
    51]])
    print(exam_scores)

    [[72 77 53]
     [86 83 90]
     [35 42 51]]

[ …
```

Figure 3.16 Creating an Array with NumPy

From the output of the array, we can see that each list in the `array()` function corresponds to one row in the array, and not one column. Furthermore, NumPy uses the outer square bracket to indicate the start and end of the array, and the inner square brackets are used to wrap the data in each row.

Due to the mechanism of the `array()` function, it is crucial to make sure what features of the data do the columns and the rows represent before creating the arrays.

2.2 Subsetting Arrays

We can use the index operator to access certain elements of an array just as indexing tuples, lists or dictionaries in Study Unit 2. And from there, we can subset an array. Nonetheless, there are two ways to subset an array using the index operator.

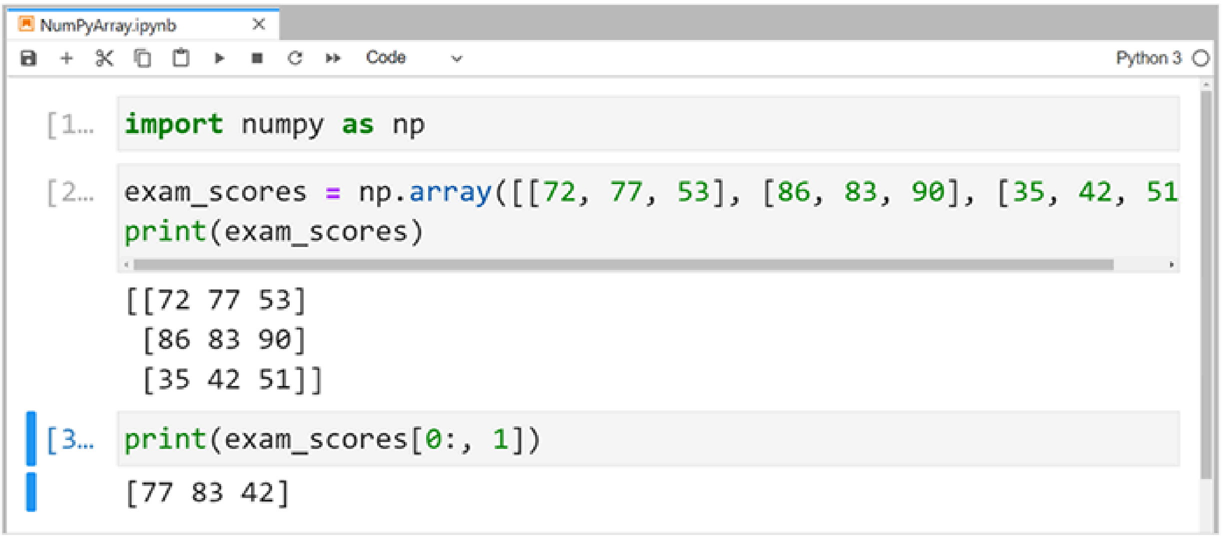
- Using index: Suppose we would like to get the second value of the first row in an array, we can extract the element by `array_name[0, 1]`, where `array_name` can be any arbitrary name of an array. The index here starts with 0, same as Python List. Recall that if we intend to do multiple indexing like `start:end`, the end index will not be included in our array subset. Furthermore, negative indexing and open-end indexing are also allowed here. Check on Chapter 1 of Study Unit 2 for more details regarding indexing.
- Using Boolean masking: Suppose we want to get all values larger than 80. A first step is to use `array_name > 80` to produce a Boolean mask. The result is a NumPy array with Boolean elements: `True` if the corresponding value is above 80, `False` if it is below. Subsequently, we can use the Boolean mask inside a pair of square brackets to do subsetting. Only those elements above 80, for which the corresponding Boolean mask is `True`, are selected. If, for instance, there are two values above 80, we will end up with a NumPy array with two values.

Since arrays can be multidimensional (ndarray), we can certainly subset every dimension of it by using multidimensional indexing. The following syntax is used for subsetting two-dimensional NumPy arrays:

```
array_name[row_index, column_index]
```

Basically, the usage of the index operator here is just like subsetting a one-dimensional Python list. The only difference is the two sets of indices in it, one for row indexing, and the other one for column indexing. The resulting subset of the array can be a single value, a row, a column, or an array with less rows and/or less columns. For multidimensional NumPy arrays, the order of the indices in the index operator must follow the sequence of the axes, or dimensions, in an array. For instance, axis one of a two-dimensional array refers to the rows and axis two to the columns.

Example (Cont'd): The array `exam_scores` created in Figure 3.16 contains data of individual students in the row and data of each subject in the column. Suppose we would now like to extract all the exam scores of the second subject.



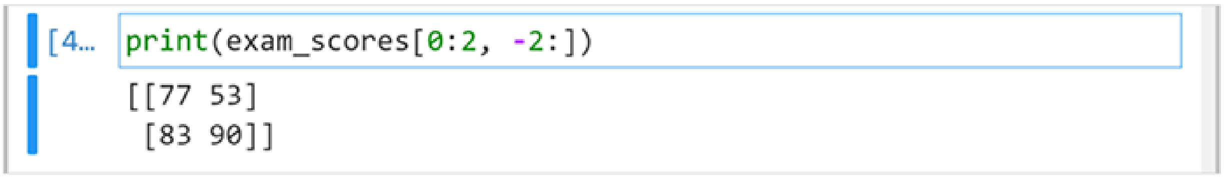
```
[1… import numpy as np

[2… exam_scores = np.array([[72, 77, 53], [86, 83, 90], [35, 42, 51
    print(exam_scores)

    [[72 77 53]
     [86 83 90]
     [35 42 51]]

[3… print(exam_scores[0:, 1])

    [77 83 42]
```

Figure 3.17 Subsetting a Column from a NumPy Array

To subset a column from a two-dimensional array, we must indicate indices for both the rows and the columns, or an error message would appear otherwise. The column index is clearly 1 here since we intend to extract the second column of the array. The row index must be multiple indexing since we would like to access the entire column.

As a result, open-end indexing starting from index 0 is the most appropriate way here to access the elements of row 1 to row 3 of column 2.

In the next step, we would like to extract the exam scores of the first two students in the last two subjects.

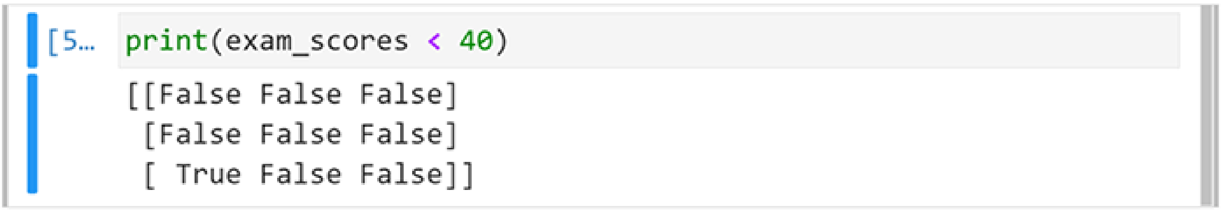


```
[4… print(exam_scores[0:2, -2:])
    [[77 53]
     [83 90]]
```

Figure 3.18 Subsetting a Sub-Array from a NumPy Array

Here, we use negative indexing for subsetting the column. Since our array has three columns, -2 is the index of the second last column. We leave the end index here open to instruct Python to “take every index until the end” which is, in this case, the last column of the array `exam_score`.

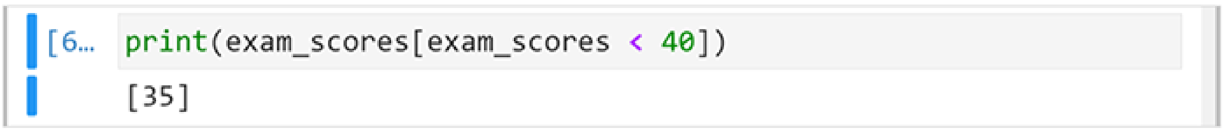
Assuming that the passing mark is 40, we would now like to subset all the failed exams. That is, we extract all exam marks below 40.



```
[5… print(exam_scores < 40)
    [[False False False]
     [False False False]
     [ True False False]]
```

Figure 3.19 Creating a Boolean Mask of a NumPy Array

If we ask Python to compare an array with a numeric value, we will obtain a Boolean mask as mentioned before. Comparing the Boolean values in Figure 3.19 with the array created in Figure 3.17, the only `True` value found here is the score of the third student in subject 1. To subset `exam_score` using the Boolean mask, we need to put the condition `exam_scores < 40` into a pair of square brackets.



```
[6… print(exam_scores[exam_scores < 40])
    [35]
```

Figure 3.20 Subsetting a NumPy Array Based on a Boolean Mask

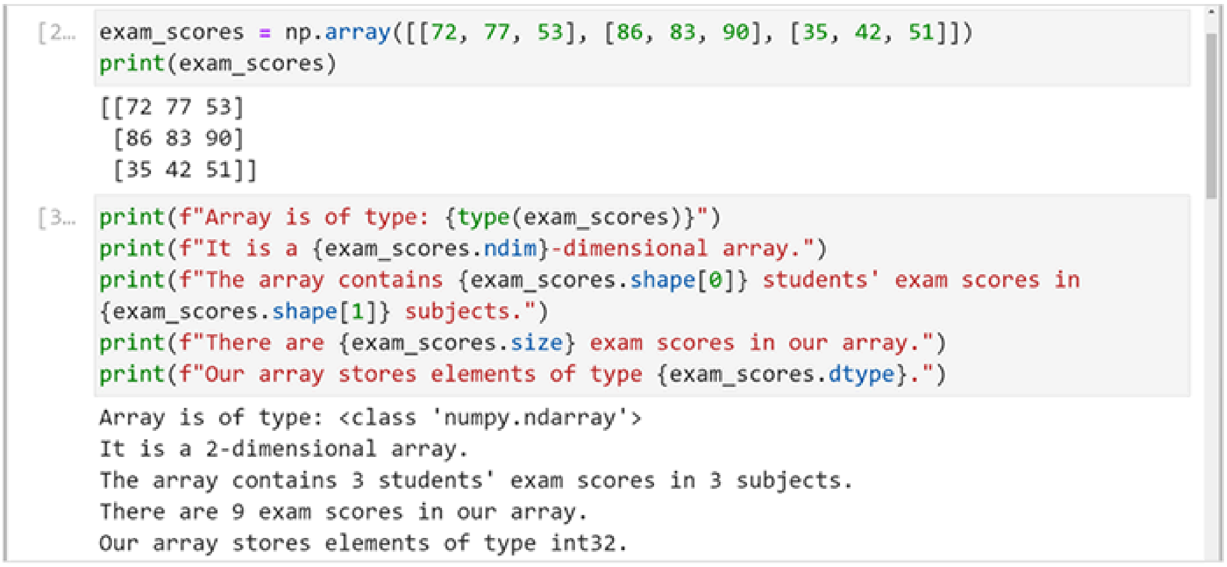
The result is the only value in `exam_scores` that is smaller than 40, namely 35, which is the score of the third student in the first subject. And it is also the only `True` value in the Boolean mask.

We can also check various properties of an ndarray using the following NumPy functions and methods:

```
type(array_name)
array_name.ndim
array_name.shape
array_name.size
array_name.dtype
```

The `type()` function indicates the type of our array, and the `.ndim` method returns the array's number of dimensions, which is usually 2 in our case. The `.shape` method provides the number of rows and columns of the given array and the `.size` method calculates the total number of elements in an array. The `.dtype` method shows us the type of data contained in the array.

Example (Cont'd): In the following, we extract all the information on the characteristics of our array exam_scores.



```
[2… exam_scores = np.array([[72, 77, 53], [86, 83, 90], [35, 42, 51]])
    print(exam_scores)

    [[72 77 53]
     [86 83 90]
     [35 42 51]]

[3… print(f"Array is of type: {type(exam_scores)}")
    print(f"It is a {exam_scores.ndim}-dimensional array.")
    print(f"The array contains {exam_scores.shape[0]} students' exam scores in
    {exam_scores.shape[1]} subjects.")
    print(f"There are {exam_scores.size} exam scores in our array.")
    print(f"Our array stores elements of type {exam_scores.dtype}.")

    Array is of type: <class 'numpy.ndarray'>
    It is a 2-dimensional array.
    The array contains 3 students' exam scores in 3 subjects.
    There are 9 exam scores in our array.
    Our array stores elements of type int32.
```

Figure 3.21 Extracting Information on the Characteristics of a NumPy Array

The array type returned from the `type()` function is a Python type output `<class 'numpy.ndarray'>`, where `ndarray` stands for n-dimensional array.

Another remarkable output is the values returned from the `.size()` method. In fact, the `.size()` method returns a tuple `(row_number, column_number)`. We subset the corresponding result in the formatted string by using the index operator `[]`.

The type of data returned by the `.dtype()` method is `int32`, a specific NumPy integer type that fixes the length of an integer variable at 32 bytes. The usual integer variable of Python has no fixed length, and its type is simply called `int`.

Read

Refer to the three links below for more details and examples on the methods "`shape`", "`ndim`" and "`size`" of NumPy arrays:

https://docs.scipy.org/doc/numpy/reference/generated/numpy.ndarray.shape.html#numpy.ndarray.shape

https://docs.scipy.org/doc/numpy/reference/generated/numpy.ndarray.ndim.html#numpy.ndarray.ndim

https://docs.scipy.org/doc/numpy/reference/generated/numpy.ndarray.size.html#numpy.ndarray.size

2.3 Working with NumPy Arrays

The NumPy package does not only facilitate the creation and management of arrays, but it also provides various functions to us to work with them.

Each function deals with specific types of variable. For instance, mathematical functions such as `log()` and `sqrt()` can only be applied on arrays with numeric values, whereas `strip()` and `upper()` are functions specifically designed for arrays with only strings in them.

Below is a table of some frequently used NumPy functions.

Table 3.1 Most Common NumPy Functions

Function	**Description**
Array Information and Operations	
`count_nonzero()`	Counts the number of non-zero values in the array

Function	Description
`extract()`	Return the elements of an array given some conditions
`nanargmin()`	Return the indices of minimum values of rows or columns ignoring missings
`nonzero()`	Return the indices of the elements that are non-zero
`partition()`	Return a partitioned copy of an array
`where()`	Return selected elements depending on condition
Statistics	
`amin(), amax()`	Returns row or column minimum or maximum
`percentile()`	Return row, column, or array percentile
`mean()`	Return row, column, or array mean
`median()`	Return row, column, or array median
`nan_to_num()`	Replace `NaN` (missings) with zero and infinity with large finite numbers
`std()`	Return row, column, or array standard deviation
`var()`	Return row, column, or array variance
Rounding	
`ceil()`	Return ceiling

Function	Description
`fix()`	Round to the nearest integer towards zero
`floor()`	Return floor
`round()`	Round to the given decimal places
`trunc()`	Return truncated value
Exponents and Logarithms	
`exp()`	Return exponential
`log()`	Return natural logarithm
`log10()`	Return base-10 logarithm
`log2()`	Return base-2 logarithm
Arithmetic	
`add()`	Add two arrays together
`absolute()`	Calculate the absolute value
`divide()`	Divide first array by second array. Non-zero elements in second array required
`mod()`	Return remainder of division
`multiply()`	Multiply two arrays with each other
`sign()`	Indicate the sign of a number

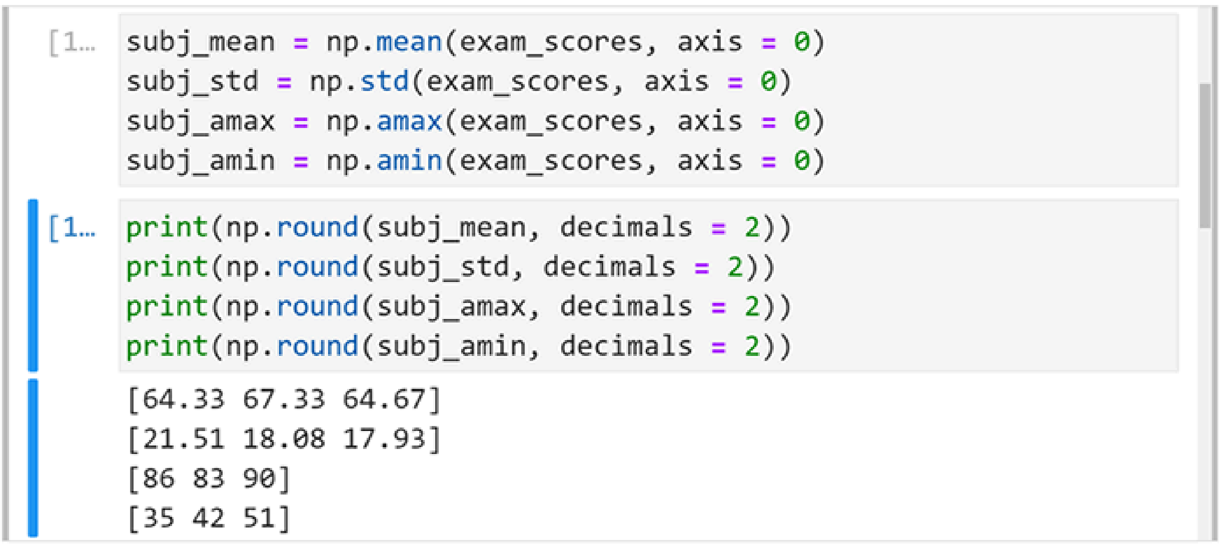
Function	Description
`sqrt()`	Return non-negative square-root
`subtract()`	Subtract two arrays from one another
Trigonometric	
`arccos()`	Return inverse cosine
`arcsin()`	Return inverse sine
`arctan()`	Return inverse tangent
`degrees()`	Convert angles from radians to degrees
`cos()`	Return cosine
`radians()`	Convert angles from degrees to radians
`sin()`	Return sine
`tan()`	Return tangent
Random Sampling	
`random.randint()`	Return an array of specified shape with random integers in a given interval
`random.random_ sample()`	Return an array of specified shape with random floats in the interval [0, 1)
`random.ranf()`	Return an array of specified shape with random floats in the interval [0, 1)

Function	**Description**
`random.normal()`	Return an array of specified shape with random floats from a normally distributed population
String Information and Operations	
`find()`	Return the lowest index of a substring in a given string
`islower()`	Check if all characters in the string are lowercase
`istitle()`	Check if a string is title-cased
`isupper()`	Check if all characters in the string are uppercase
`startswith()`	Check if a string starts with a given prefix
String Information and Operations	
`capitalize()`	Convert the first character of a string to uppercase
`join()`	Join a sequence of elements to a single string by given string separator
`lower()`	Covert a given string to lowercase
`strip()`	Remove all leading and trailing spaces from a string
`title()`	Convert the first character in each word of a string to uppercase

Function	Description
`upper()`	Covert a given string to uppercase

We must also be aware of whether the effect of a function is elementwise, row wise, column wise, or array wise. Having elementwise effects means that the function will be applied to all elements of an array individually. Having effects on array means that the function will be operating with all the array elements at the same time. Row wise and column wise effects indicate that the function will take all the elements from the same row or column into its operation. If an array has more than two dimensions, the row wise and column wise effects will become axes effects. That is, the functions can be applied in each direction of the array.

Example (Cont'd): Suppose we would like to compute some statistics of each subject's exam scores such as the mean, standard deviation, maximum, minimum, etc. We will also have to round off all these statistics to 2 decimal digits.



```
[1… subj_mean = np.mean(exam_scores, axis = 0)
    subj_std = np.std(exam_scores, axis = 0)
    subj_amax = np.amax(exam_scores, axis = 0)
    subj_amin = np.amin(exam_scores, axis = 0)
```

```
[1… print(np.round(subj_mean, decimals = 2))
    print(np.round(subj_std, decimals = 2))
    print(np.round(subj_amax, decimals = 2))
    print(np.round(subj_amin, decimals = 2))
```

```
[64.33 67.33 64.67]
[21.51 18.08 17.93]
[86 83 90]
[35 42 51]
```

Figure 3.22 Calculate Column Statistics of a NumPy Array

For the statistical functions in NumPy, we usually need to specify the `axis` argument in it. If `axis = 0`, the column statistics will be calculated. If, however, `axis = 1`, the row statistics will be returned to us by the functions. Since we intend to compute the statistics for each subject's data, which are recorded in the columns of `exam_scores`, we shall specify `axis = 0` in this case.

For the `round()` function, we specify the argument named `decimals` to 2. Recall from Chapter 3 of Study Unit 2 that it is a good programming habit to specify the argument names of a function explicitly when assigning a value to it. It is therefore important to always check the available arguments of a function carefully before using it in the program.

Read

Refer to the three links below for more details and examples on the functions `mean()`, `median()` and `std()` of the NumPy package:

https://docs.scipy.org/doc/numpy/reference/generated/numpy.mean.html#numpy.mean

https://docs.scipy.org/doc/numpy/reference/generated/numpy.median.html#numpy.median

https://docs.scipy.org/doc/numpy/reference/generated/numpy.std.html#numpy.std

Refer to the link below for more details and examples of functions provided by the NumPy package:

https://www.geeksforgeeks.org/numpy-ndarray/

Chapter 3: Plotting with Matplotlib

Lesson Recording

Plotting with matplotlib

3.1 Basic Plotting with Matplotlib

In this chapter, we will introduce some basic features of data visualisation in Python. The most common visualisation package here is “matplotlib”. Its sub-package named “pyplot” contains all the functions that we need in this Chapter.

To import the sub-package “matplotlib.pyplot”, we can use the following syntax:

```
import matplotlib.pyplot as plt
```

We use the alias “plt” here, which is also commonly found in literatures and websites, since it is short and has a clear reference to “pyplot”. Note that the above syntax has only instructed Python to import the sub-package ”pyplot”. All the other functions and sub-packages of matplotlib are not imported.

The sub-package “matplotlib.pyplot” provides many different plot types. In Chapter 3.2, we will discuss the most common ones: histogram, bar charts and scatter plots. We can customise our plots by changing colours, shapes, labels, axes, etc. according to our own needs and taste. In the following, we will introduce some basic plotting techniques based on a line plot.

The following syntax creates a simple line plot:

```
plt.plot(x, y, color, linestyle, linewidth, marker,
         markerfacecolor, markeredgecolor, markersize)
```

The argument `x` is a list of x-axis value. Correspondingly, `y` is a list of the y-axis value. The argument `color` is a string to indicate the colour of the line to be plotted such as "blue", "red", etc. The arguments `linestyle` and `linewidth` control whether the line should be solid, dashed, dotted, etc., and how thin or thick it should be. We can also choose the style of the marker of the data points on the line chart such as point, circle, square, etc. with the `marker` argument. We can also fix the colour and size of the marker with the arguments `markerfacecolor`, `markeredgecolor` and `markersize`. There are actually more arguments available for the `plot()` function. You can refer to https://matplotlib.org/api/_as_gen/matplotlib.pyplot.plot.html for all available functions of "matplotlib".

To label the axes, we can use the `xlabel()` and `ylabel()` functions.

```
plt.xlabel("My X-Label String")
plt.ylabel("My Y-Label String")
```

We can also set a title to the current plot using the `title()` function:

```
plt.title("My Plot Title")
```

Another useful plot customisation is to define the text and location of the labels on each tick of the x-axis and y-axis.

```
plt.xticks(ticks, labels, rotation)
plt.yticks(ticks, labels, rotation)
```

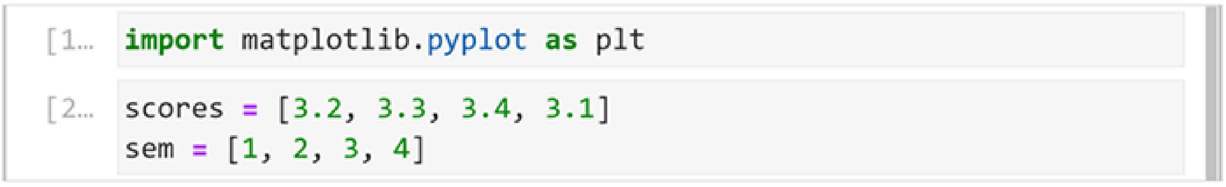
A list of labels assigned to the argument `labels` will be plotted on the locations defined with the argument `ticks`. We can also rotate the labels in case they fit optically better to the plot if they are slanted. A numeric value representing the degree of rotation can be assigned to the argument `rotation`.

Python will wait for the `show()` function to actually display all figures.

```
plt.show()
```

Note: Since our programs in this study unit are constructed and run in JupyterLab, the plots will anyway be displayed if we put all the syntaxes for plotting in one cell so that they can be executed in the same run. As a result, we will not actually need the `show()` function in the last step. However, this function is the final instruction to display the figures if we run the plotting syntaxes in the original Python program.

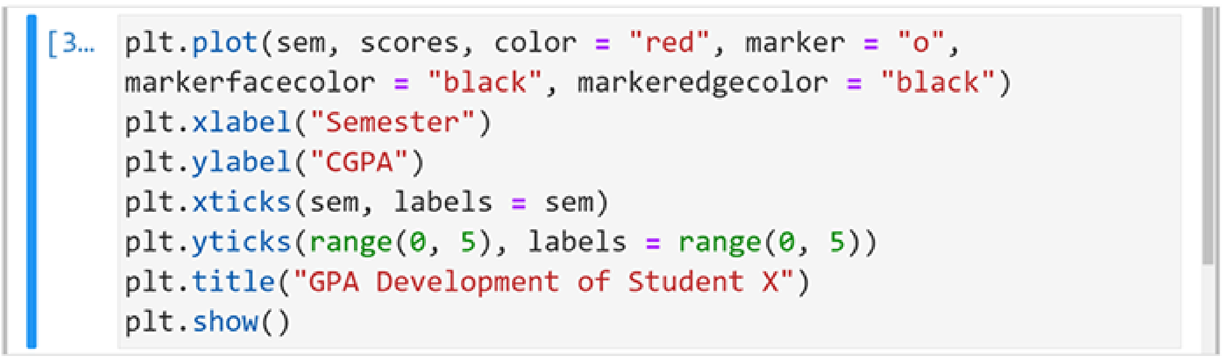
Example (cont'd): We would like to plot the CGPA development of a student in the last 4 semesters. The data are `[3.2, 3.3, 3.4, 3.1]`.



```
[1… import matplotlib.pyplot as plt

[2… scores = [3.2, 3.3, 3.4, 3.1]
    sem = [1, 2, 3, 4]
```

Figure 3.23 Importing "matplotlib.pyplot" into the Program

The following program will generate the line chart required.



```
[3… plt.plot(sem, scores, color = "red", marker = "o",
    markerfacecolor = "black", markeredgecolor = "black")
    plt.xlabel("Semester")
    plt.ylabel("CGPA")
    plt.xticks(sem, labels = sem)
    plt.yticks(range(0, 5), labels = range(0, 5))
    plt.title("GPA Development of Student X")
    plt.show()
```

Figure 3.24 Program for Creating a Line Plot

In the first line, we instruct Python to create a line chart with a red line and black circle markers. The data of the x-axis should be the semester number and the values on the y-axis are the CGPA. Correspondingly, we name the axes "Semester" and "CGPA", respectively. The location of the ticks' labels on the x-axis must be 1, 2, 3, and 4 because these are the only data of the x-axis. Since the CGPA usually lies within the interval of 0 and 4, we can create an integer list from 0 to 4 as our ticks' labels on the y-axis. We know from Chapter 5 of Study Unit 1 that we can use the `range()` function to generate such a list. Here, we integrate it within the `plt.yticks()` function. In the final step, we add a title to the line plot to highlight the topic of our chart using the `title()` function. With the `plt.show()` function, we let Python generate the plot.

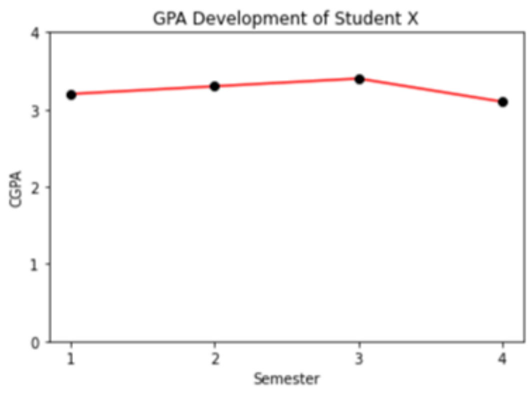


Figure 3.25 Line Plot Generated by "matplotlib.pyplot"

Read

Refer to the three links below for more details and examples on the functions `plot()` of the "matplotlib.pyplot" package:

https://matplotlib.org/api/_as_gen/matplotlib.pyplot.plot.html

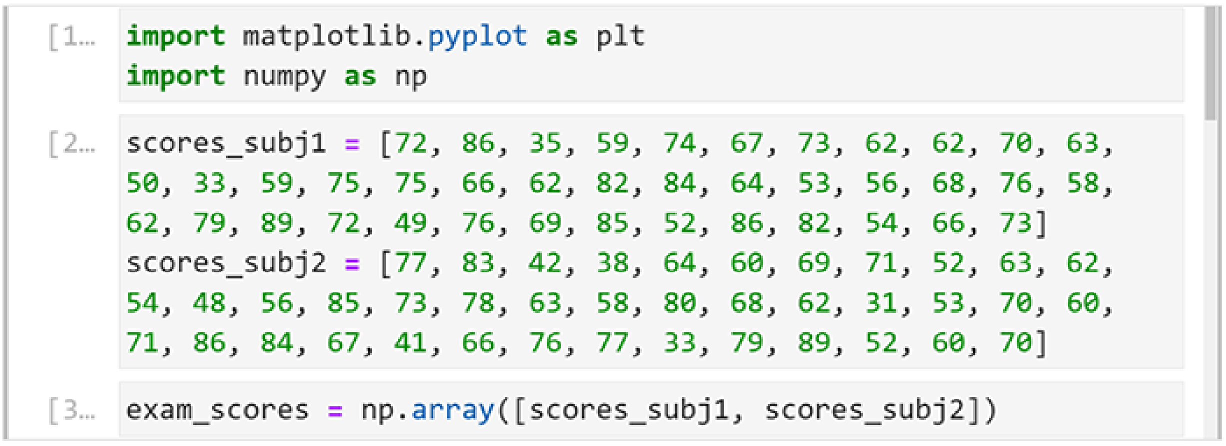
3.2 Histograms and Scatter Plot

The histogram is another common type of plots in data analytics. It shows the distribution of a variable by plotting the frequencies that certain ranges of value occur in a sample.

```
plt.hist(x, bins = None, range = None, align = "mid",
         orientation = "vertical", rwidth = None, color =
 None)
```

The `hist()` function has many arguments to control the histogram layout. The above introduction of the function only includes the most common ones. For instance, we can decide how many `bins` (bars) and which `range` of the values it should contain. We can also choose to have a histogram with horizontal bars by changing the `orientation` argument. With the arguments such as `rwidth`, which represents the width of the bars, `align`, with which we can position the bars between two ticks or on top of a tick, and `color`, we can format the bars according to our needs.

Example (cont'd): Suppose the exam scores of the two subjects (taken by the same students) are now completely available, and we would like to generate a histogram for subject 1 to look at the distribution.

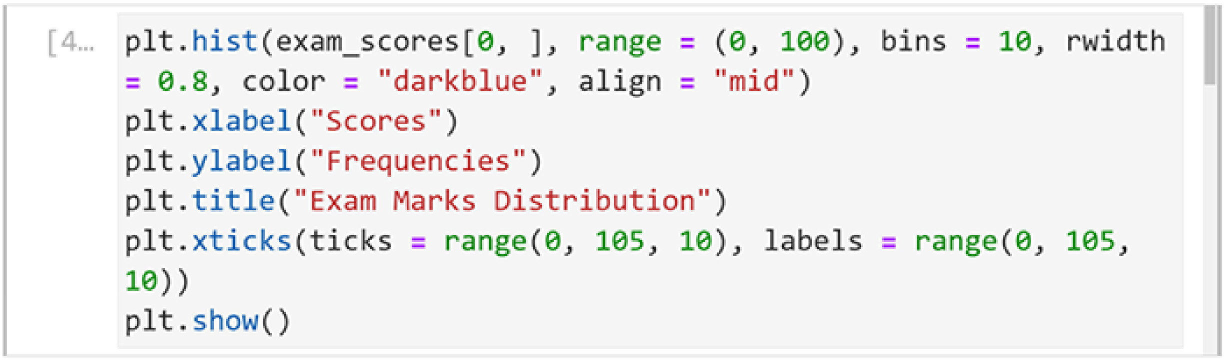


```
[1… import matplotlib.pyplot as plt
    import numpy as np

[2… scores_subj1 = [72, 86, 35, 59, 74, 67, 73, 62, 62, 70, 63,
    50, 33, 59, 75, 75, 66, 62, 82, 84, 64, 53, 56, 68, 76, 58,
    62, 79, 89, 72, 49, 76, 69, 85, 52, 86, 82, 54, 66, 73]
    scores_subj2 = [77, 83, 42, 38, 64, 60, 69, 71, 52, 63, 62,
    54, 48, 56, 85, 73, 78, 63, 58, 80, 68, 62, 31, 53, 70, 60,
    71, 86, 84, 67, 41, 66, 76, 77, 33, 79, 89, 52, 60, 70]

[3… exam_scores = np.array([scores_subj1, scores_subj2])
```

Figure 3.26 Creating Data Array for Subsequent Plotting

First, we import both the "NumPy" and "matplotlib" packages, and then we create two lists with the exam scores of each of the subjects. Eventually, we create a NumPy array that contains both lists as its elements.

After that, the following program is written to generate the histogram.



```
[4… plt.hist(exam_scores[0, ], range = (0, 100), bins = 10, rwidth
    = 0.8, color = "darkblue", align = "mid")
    plt.xlabel("Scores")
    plt.ylabel("Frequencies")
    plt.title("Exam Marks Distribution")
    plt.xticks(ticks = range(0, 105, 10), labels = range(0, 105,
    10))
    plt.show()
```

Figure 3.27 Program for Creating a Histogram

In the first line, we instruct Python to create a histogram based on the scores of the first subject's examination, which are stored in the first row of the array `exam_scores`. We set the `range` argument to be between 0 and 100 to ensure that extreme categories such as 0-10 or 90-100 marks are also included in the chart although their frequencies could be 0. The number of bins is fixed at 10 here so that we gain an accurate image of the distribution. The width of the bars is reduced from 1 to 0.8 so that they are not touching each other, and they are placed between two ticks on the x-axis (`align = "mid"`) to indicate the score range each bar represents.

The axes are named “Scores” and “Frequencies”, according to their nature. A title is also given to the histogram called “Exam Marks Distribution”. The ticks on the x-axis are placed with a gap of 10 marks between 0 and 100. We extended the range to 105 since the right end is not included by the `range()` function.

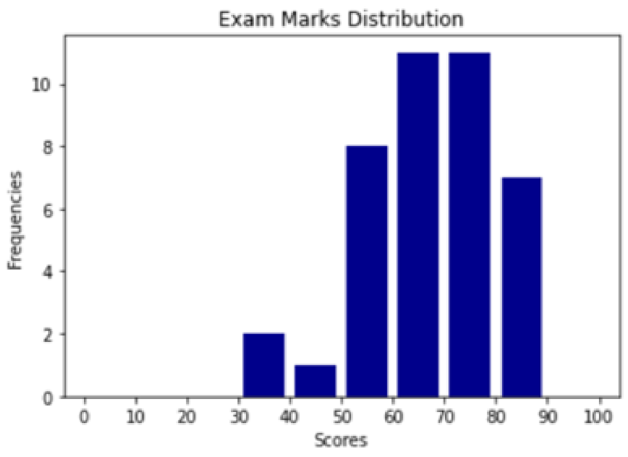


Figure 3.28 Historam Generated by “matplotlib.pyplot”

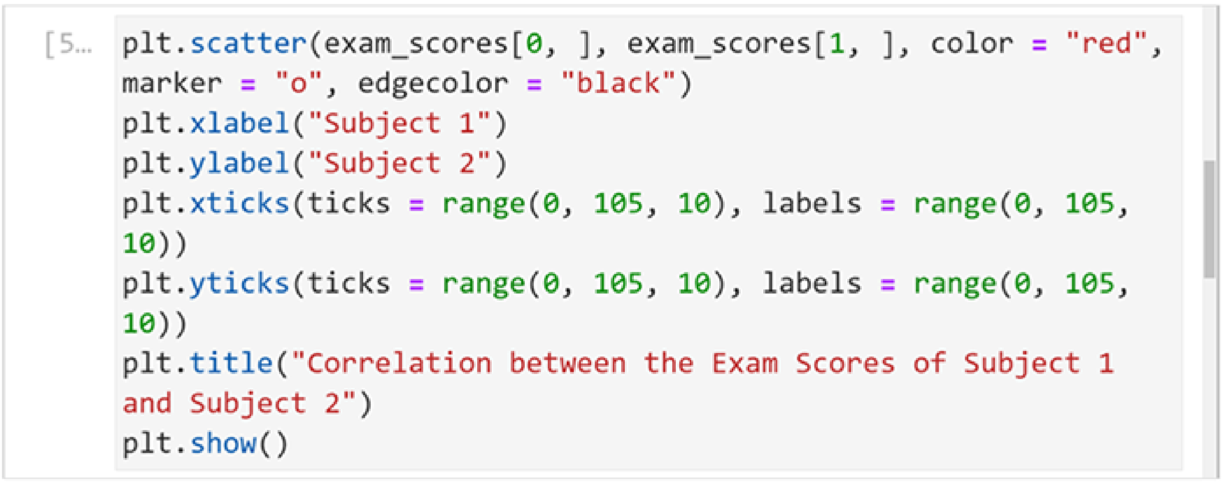
Scatter plots are often used to study the relationship between two variables, which is usually referred as their correlation. The values of the first variable are plotted in the x-axis and the values of the second variable in the y-axis. If the data dots are scatted around the 45 degrees line of the chart, we can conclude that these variables are correlated with each other.

To create a scatter plot, “matplotlib.pyplot” provides the following possibility:

```
plt.scatter(x, y, color = None, marker = None,
            linewidths = None, edgecolors = None)
```

Same as the hist() function, we only list out some of the most common arguments of the scatter() function here. For instance, we can change the colour and style of the markers, assign another colour to the markers' edge, and adjust its width for more sophisticated visualisation. You can refer to https://matplotlib.org/api/_as_gen/matplotlib.pyplot.scatter.html to find out more available arguments to control the scatter plot layout.

Example (cont'd): Now we investigate the correlation between the students' performances in the two exams. The same set of data as in Figure 3.26 is used here to generate a scatter plot by the following program.



```
[5… plt.scatter(exam_scores[0, ], exam_scores[1, ], color = "red",
    marker = "o", edgecolor = "black")
    plt.xlabel("Subject 1")
    plt.ylabel("Subject 2")
    plt.xticks(ticks = range(0, 105, 10), labels = range(0, 105,
    10))
    plt.yticks(ticks = range(0, 105, 10), labels = range(0, 105,
    10))
    plt.title("Correlation between the Exam Scores of Subject 1
    and Subject 2")
    plt.show()
```

Figure 3.29 Program for Creating a Scatter Plot

First, we instruct Python to create a scatter plot based on the array exam_scores. The first row of the array contains the data of the x-axis, and the data in the second row are values of the y-axis. The markers should be red circles and have black edges. The axes are also named accordingly: "Subject 1" for the x-axis and "Subject 2" for the y-axis. The title of the scatter plot is "Correlation between the Exam Scores of Subject 1 and Subject 2". The ticks on the x-axis and y-axis are placed with a gap of 10 marks between 0 and 100. We extended the range to 105 to include 100 in our chart since the right end is not included by the range() function.

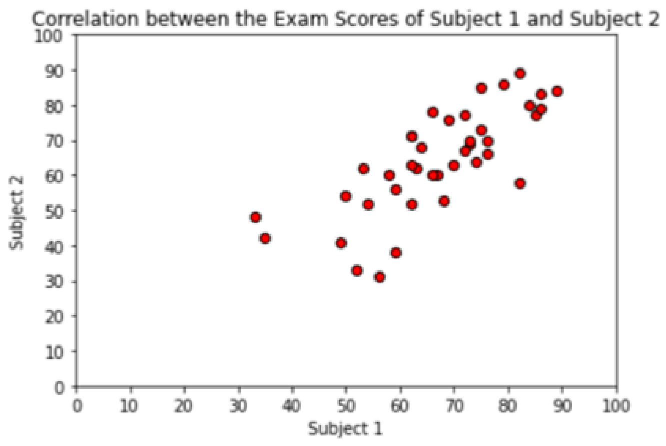


Figure 3.30 Scatter Plot Generated by "matplotlib.pyplot"

Read

Refer to the link below for more details and examples on the `hist()` function of the "matplotlib.pyplot" package:

https://matplotlib.org/api/_as_gen/matplotlib.pyplot.hist.html

Refer to the link below for more details and examples on the `scatter()` function of the "matplotlib.pyplot" package:

https://matplotlib.org/api/_as_gen/matplotlib.pyplot.scatter.html

Summary

We have discussed two Python packages in this study unit: NumPy and matplotlib. They are the fundamental packages for efficient scientific computing and data visualisation with Python, respectively. We have learned the basics of the two packages such as subsetting and some functions to operate on NumPy arrays, and some functions of matplotlib for plotting and customising basic charts for analytics such as line chart, histogram and scatter plot.

Formative Assessment

1. What is the use of a Markdown cell in JupyterLab?

    a. It runs only programs in which the matplotlib package is involved.

    b. It has more advanced functionalities than a usual Python3 code cell.

    c. It is used to embed elaborative texts to the program.

    d. It can only be used if a Python function is not compatible with Python3.

2. What is the output of the following program?

```
a = np.array([1, 2, -1], [0, 3, -2])
print(a)
```

a.
```
[[1, 2, -1]
[0, 3, -2]]
```

b. `[1, 2, -1, 0, 3, -2]`

c. `[[1, 2, -1], [0, 3, -2]]`

d. Error message

3. What is the output of the following program?

```
a = np.array([[1, 2, -1], [0, 3, -2]])
print(a[a < 0])
```

a. `[-1 -2]`

b.
```
[[-1]
 [-2]]
```

c. `[[], [-1, -2]]`

d. `[[, , -1], [, , -2]]`

4. Which values will remain in the output based on the following code?

```
a = np.array([[1, 2, -1], [0, 3, -2]])
print(a[1:, -2:-1])
```

a. `[[0, 3, -2]]`

b. `[[3, -2]]`

c. `[[3]]`

d. `[[1, -1]]`

5. What information does the NumPy method `.shape` provide?

a. The dimension number of an array

b. The number of rows and columns of an array

c. The total number of elements of an array

d. The type of data in an array

6. Which of the following NumPy functions does not have elementwise effects?

a. `cos()`

b. `exp()`

c. `fix()`

d. `var()`

7. Which of the following is not an argument of the `plt.plot()` function?

a. `range`

b. `marker`

c. `color`

   d. `linestyle`

8. What does the `rotation` argument of the `plt.xticks()` function control?
   a. It controls the rotation of the plot.
   b. It controls the rotation of the labels of the ticks.
   c. It controls the rotation of the axis labels.
   d. It controls the rotation of the main title.

9. What does the function `plt.show()` actually do?
   a. It displays the most previous command of “matplotlib.pyplot”.
   b. It sends all the figures to the connected printer for printing.
   c. It displays all the figures to the screen.
   d. It shows all the available arguments of the most previously executed function.

10. Where will the bars of a histogram be placed if we set `align = "left"`?
    a. On top of the lower boundary tick
    b. On top of the upper boundary tick
    c. Between the ticks of the upper and lower boundaries
    d. To the left of the y-axis label

Solutions or Suggested Answers

Formative Assessment

1. What is the use of a Markdown cell in JupyterLab?

   a. It runs only programs in which the matplotlib package is involved.

   Incorrect. JupyterLab can deal with all Python packages and you can run it in any arbitrary code cell.

   b. It has more advanced functionalities than a usual Python3 code cell.

   Incorrect. Markdown cells do not have any Python functionality.

   c. It is used to embed elaborative texts to the program.

   Correct. We can write elaborative texts to the program or HTML code to design a website in Markdown cells.

   d. It can only be used if a Python function is not compatible with Python3.

   Incorrect. Markdown cells are not used for programming purpose. It does not matter whether a Python function is compatible with Python3 or not.

2. What is the output of the following program?

```
a = np.array([1, 2, -1], [0, 3, -2])
print(a)
```

   a.

```
[[1, 2, -1]
[0, 3, -2]]
```

Incorrect. Since a pair of square brackets to wrap up both the lists inside the `array()` function is missing, it is not a valid program.

b. `[1, 2, -1, 0, 3, -2]`

Incorrect. Since the user intends to create an array with two rows, the result cannot be a Python list with all the elements in it.

c. `[[1, 2, -1], [0, 3, -2]]`

Incorrect. The user intends to create an array with two rows, the result cannot be a Python list with two sub-lists in it.

d. Error message

Correct. Since a pair of square brackets to wrap up both the lists inside the `array()` function is missing, it is not a valid program and an error message will appear.

3. What is the output of the following program?

```
a = np.array([[1, 2, -1], [0, 3, -2]])
print(a[a < 0])
```

a. `[-1 -2]`

Correct. The subsetting result in NumPy will be presented in a one-dimensional row array.

b.
```
[[-1]
 [-2]]
```

Incorrect. The subsetting result in NumPy will not be presented as a column array.

c. `[[], [-1, -2]]`

Incorrect. The subsetting result in NumPy will not be presented in a multidimensional array.

d. `[[, , -1], [, , -2]]`

Incorrect. The subsetting result in NumPy will not be presented in a multidimensional array and positions where the value was False in the Boolean mask will not be kept in the resulting array.

4. Which values will remain in the output based on the following code?

```
a = np.array([[1, 2, -1], [0, 3, -2]])
print(a[1:, -2:-1])
```

a. `[[0, 3, -2]]`

Incorrect. The only column index in this subsetting is -2, the second last column of the array. As a result, we cannot have three values remaining in the output.

b. `[[3, -2]]`

Incorrect. The only column index in this subsetting is -2, the second last column of the array. As a result, we cannot have two values remaining in the output.

c. `[[3]]`

Correct. The only column index in this subsetting is -2, the second last column of the array. Since the row index is 1, indicating the second row

of the array, the output here should be the value positioning in the second column of the second row, which is 3.

d. `[[1, -1]]`

Incorrect. Since the row index is 1, indicating the second row of the array, the output here cannot contain any value of the first row.

5. What information does the NumPy method `.shape` provide?

a. The dimension number of an array

Incorrect. The dimension number of an array can be extracted by the `.ndim` method.

b. The number of rows and columns of an array

Correct. The `.shape` method returns the number of rows and the number of columns as a tuple with two elements.

c. The total number of elements of an array

Incorrect. The total number of elements of an array can be extracted by the `.size` method.

d. The type of data in an array

Incorrect. The type of data in an array can be extracted by the `.dtype` method.

6. Which of the following NumPy functions does not have elementwise effects?

a. `cos()`

Incorrect. The `cos()` function calculates the cosine of each element of a numeric array.

b. `exp()`

Incorrect. The `exp()` function calculates the exponential of each element of a numeric array.

c. `fix()`

Incorrect. The `fix()` function rounds each element of a numeric array to the nearest integer towards zero.

d. `var()`

Correct. The `var()` function calculates the variance of each row, each column, or the entire array.

7. Which of the following is not an argument of the `plt.plot()` function?

a. `range`

Correct. The `plt.plot()` function does not have the range argument.

b. `marker`

Incorrect. The `marker` argument controls the marker style of a line plot.

c. `color`

Incorrect. The `color` argument controls the line colour of a line plot.

d. `linestyle`

Incorrect. The `linestyle` argument controls the line style of a line plot.

8. What does the `rotation` argument of the `plt.xticks()` function control?

a. It controls the rotation of the plot.

Incorrect. The plot cannot be rotated generally. For some types of plot such as histogram or bar chart, their bars can be presented vertically or horizontally. But the plot cannot be rotated completely.

b. It controls the rotation of the labels of the ticks.

   Correct. We can rotate the labels of the ticks if we want them slanted.

c. It controls the rotation of the axis labels.

   Incorrect. It does not control the layout of the axis labels at all.

d. It controls the rotation of the main title.

   Incorrect. It does not control the layout of the main title at all.

9. What does the function `plt.show()` actually do?

   a. It displays the most previous command of “matplotlib.pyplot”.

      Incorrect. It does not only display the most previous command of “matplotlib.pyplot”.

   b. It sends all the figures to the connected printer for printing.

      Incorrect. It does not send anything to the printer for printing.

   c. It displays all the figures to the screen.

      Correct. It will display all the figures to the screen.

   d. It shows all the available arguments of the most previously executed function.

      Incorrect. It is not a function to print out the arguments of a function at all.

10. Where will the bars of a histogram be placed if we set `align = "left"`?

    a. On top of the lower boundary tick

       Correct. They will be placed on top of the lower boundary tick.

    b. On top of the upper boundary tick

Incorrect. If they should be placed on top of the upper boundary tick, we should have set `align = “right”`.

c. Between the ticks of the upper and lower boundaries

Incorrect. If they should be placed between the ticks of the upper and lower boundaries, we should have set `align = “mid”`.

d. To the left of the y-axis label

Incorrect. The align argument only controls how the histogram bars are placed in relation to the ticks on the x-axis.

References

GeeksforGeeks. (n.d.). *N-Dimensional array(ndarray) in numpy*. https://www.geeksforgeeks.org/numpy-ndarray/

JupyterLab stable. (n.d.). *JupyterLab documentation*. Project Jupyter. https://jupyterlab.readthedocs.io/en/stable/

NumPy. (2020, Jun 29). *numpy.mean*. The SciPy Community. https://docs.scipy.org/doc/numpy/reference/generated/numpy.mean.html#numpy.mean

NumPy. (2020, Jun 29). *numpy.median*. The SciPy Community. https://docs.scipy.org/doc/numpy/reference/generated/numpy.median.html#numpy.median

NumPy. (2020, Jun 29). *numpy.ndarray.ndim*. The SciPy Community. https://docs.scipy.org/doc/numpy/reference/generated/numpy.ndarray.ndim.html#numpy.ndarray.ndim

NumPy. (2020, Jun 29). *numpy.ndarray.shape*. The SciPy Community. https://docs.scipy.org/doc/numpy/reference/generated/numpy.ndarray.shape.html#numpy.ndarray.shape

NumPy. (2020, Jun 29). *numpy.ndarray.size*. The SciPy Community. https://docs.scipy.org/doc/numpy/reference/generated/numpy.ndarray.size.html#numpy.ndarray.size

NumPy. (2020, Jun 29). *numpy.std*. The SciPy Community. https://docs.scipy.org/doc/numpy/reference/generated/numpy.std.html#numpy.std

matplotlib. (n.d.). *matplotlib.plot.scatter*. The Matplotlib development team. https://matplotlib.org/api/_as_gen/matplotlib.pyplot.scatter.html

matplotlib. (n.d.). *matplotlib.pyplot.plot*. The Matplotlib development team. https://matplotlib.org/api/_as_gen/matplotlib.pyplot.plot.html

matplotlib. (n.d.). *matplotlib.pyploy.hist*. The Matplotlib development team. https://matplotlib.org/api/_as_gen/matplotlib.pyplot.hist.html

Project Jupyter. (2021, Jan 06). *Home*. https://jupyter.org/

Study Unit 4

Data Management

Learning Outcomes

By the end of this unit, you should be able to:

1. Explain the operations on datasets
2. Prepare data for analysis using Python programming

Overview

This unit will introduce the key data structure for analytics in Python: the pandas DataFrame. We will learn to develop Python programs to import data from external sources and convert them to DataFrames, and then to index and query these structures. We will then deepen our understanding of the pandas package by learning its efficient functionality on merging multiple DataFrames, identifying and dealing with missing data and outliers, sorting, grouping and transforming data, as well as discretising numeric variables to bins.

Chapter 1: Import Data

Lesson Recording

Import Data in pandas

In the previous study units, we have learned the basic techniques of Python programming. In this and the next study units, we will discuss in detail how Python can be used for data management and data analytics.

The most common package for data management in Python is “pandas”. After installing pandas using pip, we can import it in our program by the following syntax:

```
import pandas as pd
```

Here, we use the alias `pd` to refer to the pandas package in our programs.

To start working with pandas, we need to have Python compatible datasets. Data circulating in organisations or on the internet are mostly saved as text files or worksheets. Text editors, spreadsheets, and data management apps are popular tools for opening and working with them. Pandas actually provides the same possibilities. The first step here is to load a dataset in the Python environment and open it in the format of pandas. Suppose we have a dataset from an external source saved as a .csv text file; we can import it by the following pandas function:

```
DataFrame_name = pd.read_csv("csv_file_name.csv")
```

The content stored in the file “`csv_file_name.csv`” will be then assigned to the pandas dataset object, or DataFrame, named `DataFrame_name`. The function `read_csv()` is called a reader since it reads in specific format of data files and converts them to pandas DataFrame.

Same as functions in NumPy or matplotlib, the `read_csv()` function has more arguments than we list out here. We can adjust the execution of the reader to the specifications of the .csv file with these arguments. For instance, we can specify the character string of the delimiter, the row number in which the header is stored, the path of the .csv file, etc. You can refer to https://pandas.pydata.org/docs/user_guide/io.html#io-read-csv-table for further details.

Since .csv is not the only common file format of data files, pandas also allows the import of other file formats such as Excel spreadsheets, SPSS data or Stata data into Python by providing the functions listed in the following table.

Table 4.1 Most Common Data File Formats and the Corresponding Reader in pandas

Reader	Format Type	Data Description
`read_csv()`	text	CSV
`read_html()`	text	HTML
`read_clipboard()`	text	Local clipboard
`read_excel()`	binary	MS Excel
`read_stata()`	binary	Stata
`read_sas()`	binary	SAS
`read_spss()`	binary	SPSS
`read_pickle()`	binary	Python Pickle Format
`read_sql()`	SQL	SQL

Reader	Format Type	Data Description
`read_gbq()`	SQL	Google BigQuery

We can then use the `.head()` method to display the first five rows of the imported dataset. It is important to check whether the data have been accurately imported.

```
DataFrame_name.head()
```

Alternatively, we can also use the conventional `print()` function to display the whole DataFrame. Nevertheless, this can be quite frustrating if the dataset contains many rows and columns, and the output does not fit to the window properly. Another way to print the whole DataFrame is to use the `display()` function or omit the function completely and simply execute a syntax with only the name of the DataFrame.

Note: When using NumPy ndarrays (n-dimensional arrays) to store multi dimensional data, a burden is placed on the programmer to specify the orientation of the dataset, because axes are considered more or less equivalent. The meanings of rows and columns of an array do not differ significantly and switching the roles of rows and columns (transpose) to store the data does not change the nature of the array at all. And the functionality of the NumPy functions will remain. For pandas DataFrame, or datasets in general, the rows record individual observations, and the columns represent the features, or variables, of the data. Their roles usually do not change throughout the entire analysis process. Thus, the axes lend more semantic meaning to the data, and hence reduce the amount of mental effort required to code up data transformation.

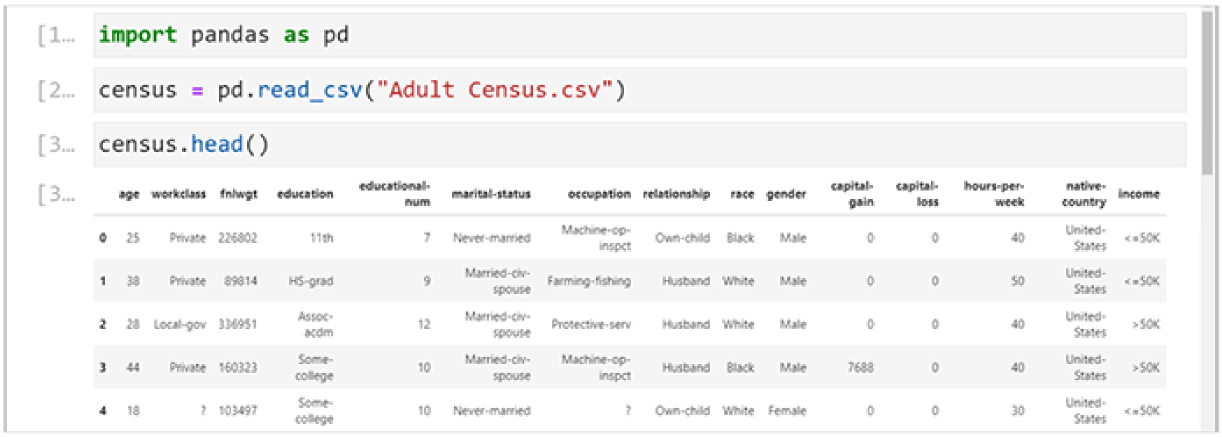
Example (Adult Census Data): The US Adult Census dataset is a repository of 48,842 entries extracted from the 1994 US Census database. The dataset is downloadable from https://www.kaggle.com/wenruliu/adult-income-dataset. It is used to predict whether income would exceed $50,000 per year according to the 14 social-demographic attributes (Source: http://www.cs.toronto.edu/~delve/data/adult/adultDetail.html). Below is a list of the available variables in the dataset:

1. **age**: the age of an individual. Its value can be any integer greater than 0.
2. **workclass**: a general term to represent the employment status of an individual. Its value can be `Private`, `Self-emp-not-inc`, `Self-emp-inc`, `Federal-gov`, `Local-gov`, `State-gov`, `Without-pay`, `Never-worked`.
3. **fnlwgt**: final weight. In other words, this is the number of people the entry represents. Its value can be any integer greater than 0.
4. **education**: the highest level of education achieved by an individual. Its value can be `Bachelors`, `Some-college`, `11th`, `HS-grad`, `Prof-school`, `Assoc-acdm`, `Assoc-voc`, `9th`, `7th-8th`, `12th`, `Masters`, `1st-4th`, `10th`, `Doctorate`, `5th-6th`, Preschool.
5. **educational-num**: the highest level of education achieved in numerical form. Its value can be any Integer greater than 0.
6. **marital-status**: marital status of an individual. Married-civ-spouse corresponds to a civilian spouse while `Married-AF-spouse` is a spouse in the Armed Forces. Its value can be `Married-civ-spouse`, `Divorced`, `Never-married`, `Separated`, `Widowed`, `Married-spouse-absent`, `Married-AF-spouse`.
7. **occupation**: the general type of occupation of an individual. Its value can be `Tech-support`, `Craft-repair`, `Other-service`, `Sales`, `Exec-managerial`, `Prof-specialty`, `Handlers-cleaners`, `Machine-op-`

inspct, Adm-clerical, Farming-fishing, Transport-moving, Priv-house-serv, Protective-serv, Armed-Forces.

8. **relationship**: represents what this individual is relative to others. For example, an individual could be a husband. Each entry only has one relationship attribute. Its value can be `Wife`, `Own-child`, `Husband`, `Not-in-family`, `Other-relative`, `Unmarried`.
9. **race**: Descriptions of an individual's race. Its value can be `White`, `Asian-Pac-Islander`, `Amer-Indian-Eskimo`, `Other`, `Black`.
10. **gender**: the biological gender of the individual. Its value can be Male, Female.
11. **capital-gain**: capital gains for an individual. Its value can be any integer greater than or equal to 0.
12. **capital-loss**: capital loss for an individual. Its value can be any integer greater than or equal to 0.
13. **hours-per-week**: the hours an individual has reported to work per week. Its value can be any positive real number or 0.
14. **native-country**: country of origin for an individual. Its value can be `United-States`, `Cambodia`, `England`, `Puerto-Rico`, `Canada`, `Germany`, `Outlying-US(Guam-USVI-etc)`, `India`, `Japan`, `Greece`, `South`, `China`, `Cuba`, `Iran`, `Honduras`, `Philippines`, `Italy`, `Poland`, `Jamaica`, `Vietnam`, `Mexico`, `Portugal`, `Ireland`, `France`, `Dominican-Republic`, `Laos`, `Ecuador`, `Taiwan`, `Haiti`, `Columbia`, `Hungary`, `Guatemala`, `Nicaragua`, `Scotland`, `Thailand`, `Yugoslavia`, `El-Salvador`, `Trinadad&Tobago`, `Peru`, `Hong`, `Holand-Netherlands`.
15. **income**: whether or not an individual makes more than $50,000 annually. Its value can be `<=50k`, `>50k`.

Since the original data file is saved in .csv format, we can use the `read_csv()` function to import the dataset to Python.



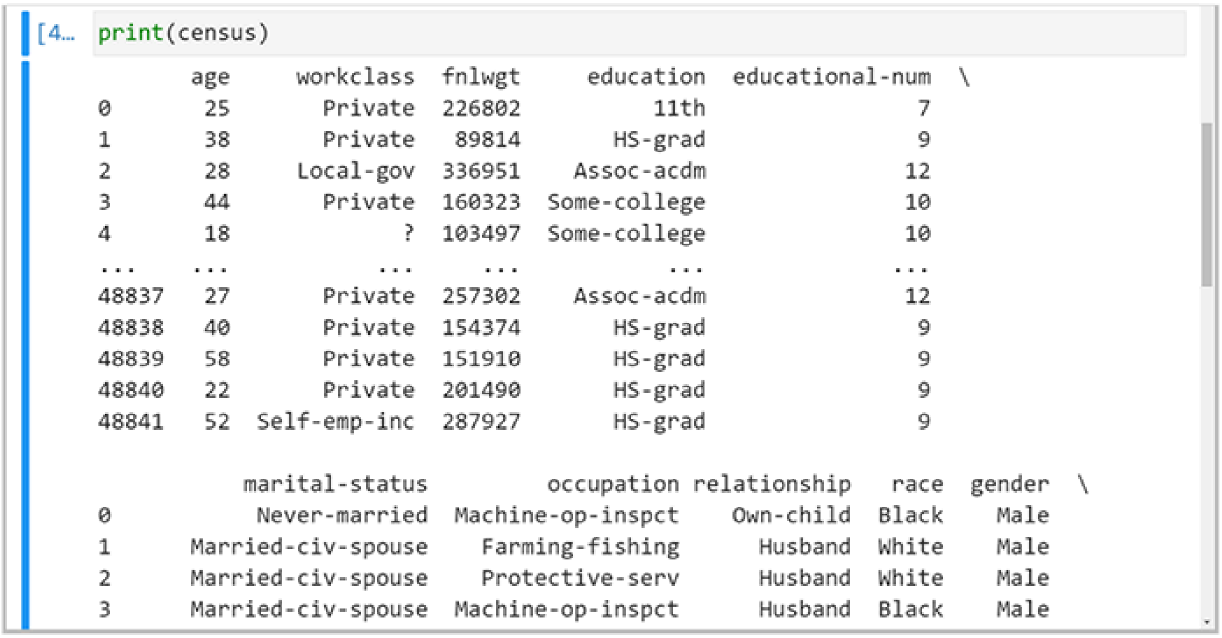
```
[1… import pandas as pd

[2… census = pd.read_csv("Adult Census.csv")

[3… census.head()
```

[3…

	age	workclass	fnlwgt	education	educational-num	marital-status	occupation	relationship	race	gender	capital-gain	capital-loss	hours-per-week	native-country	income
0	25	Private	226802	11th	7	Never-married	Machine-op-inspct	Own-child	Black	Male	0	0	40	United-States	<=50K
1	38	Private	89814	HS-grad	9	Married-civ-spouse	Farming-fishing	Husband	White	Male	0	0	50	United-States	<=50K
2	28	Local-gov	336951	Assoc-acdm	12	Married-civ-spouse	Protective-serv	Husband	White	Male	0	0	40	United-States	>50K
3	44	Private	160323	Some-college	10	Married-civ-spouse	Machine-op-inspct	Husband	Black	Male	7688	0	40	United-States	>50K
4	18	?	103497	Some-college	10	Never-married	?	Own-child	White	Female	0	0	30	United-States	<=50K

Figure 4.1 Importing Data with pandas

If we printed the census dataset using the `print()` function, the output would not fit the window at all.

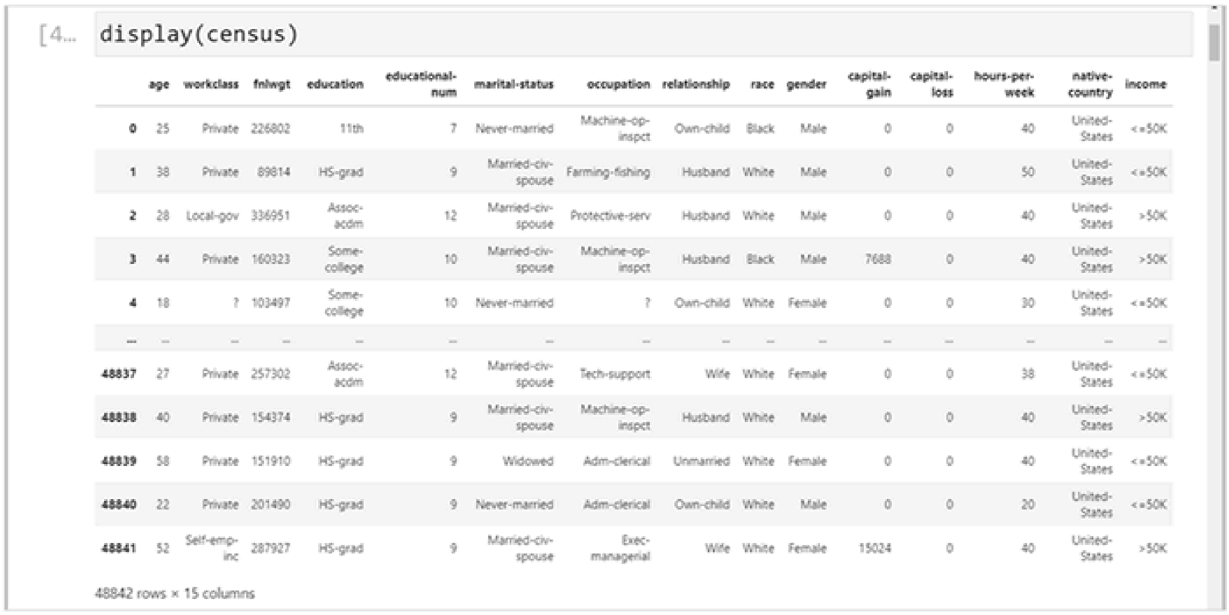


```
[4… print(census)
       age     workclass  fnlwgt     education  educational-num  \
0       25       Private  226802          11th                7
1       38       Private   89814       HS-grad                9
2       28     Local-gov  336951    Assoc-acdm               12
3       44       Private  160323  Some-college               10
4       18             ?  103497  Some-college               10
...    ...           ...     ...           ...              ...
48837   27       Private  257302    Assoc-acdm               12
48838   40       Private  154374       HS-grad                9
48839   58       Private  151910       HS-grad                9
48840   22       Private  201490       HS-grad                9
48841   52  Self-emp-inc  287927       HS-grad                9

           marital-status         occupation relationship   race  gender  \
0           Never-married  Machine-op-inspct    Own-child  Black    Male
1      Married-civ-spouse    Farming-fishing      Husband  White    Male
2      Married-civ-spouse    Protective-serv      Husband  White    Male
3      Married-civ-spouse  Machine-op-inspct      Husband  Black    Male
```

Figure 4.2 Printing Entire Imported Dataset

As shown in Figure 4.2, the DataFrame will be truncated anyway. Eventually, only the head and the tail of the dataset will be displayed due to lack of space. As a result, it is more advisable to use the `pd.head()` than `print()` for control purpose.

Furthermore, Python provides the `display()` function.



```
[4… display(census)
```

	age	workclass	fnlwgt	education	educational-num	marital-status	occupation	relationship	race	gender	capital-gain	capital-loss	hours-per-week	native-country	income
0	25	Private	226802	11th	7	Never-married	Machine-op-inspct	Own-child	Black	Male	0	0	40	United-States	<=50K
1	38	Private	89814	HS-grad	9	Married-civ-spouse	Farming-fishing	Husband	White	Male	0	0	50	United-States	<=50K
2	28	Local-gov	336951	Assoc-acdm	12	Married-civ-spouse	Protective-serv	Husband	White	Male	0	0	40	United-States	>50K
3	44	Private	160323	Some-college	10	Married-civ-spouse	Machine-op-inspct	Husband	Black	Male	7688	0	40	United-States	>50K
4	18	?	103497	Some-college	10	Never-married	?	Own-child	White	Female	0	0	30	United-States	<=50K
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
48837	27	Private	257302	Assoc-acdm	12	Married-civ-spouse	Tech-support	Wife	White	Female	0	0	38	United-States	<=50K
48838	40	Private	154374	HS-grad	9	Married-civ-spouse	Machine-op-inspct	Husband	White	Male	0	0	40	United-States	>50K
48839	58	Private	151910	HS-grad	9	Widowed	Adm-clerical	Unmarried	White	Female	0	0	40	United-States	<=50K
48840	22	Private	201490	HS-grad	9	Never-married	Adm-clerical	Own-child	White	Male	0	0	20	United-States	<=50K
48841	52	Self-emp-inc	287927	HS-grad	9	Married-civ-spouse	Exec-managerial	Wife	White	Female	15024	0	40	United-States	>50K

48842 rows × 15 columns

Figure 4.3 Printing the Head and Tail of the Imported Dataset by `display()`

The `display()` function prints out the first 5 and last 5 observations of the DataFrame. The font is smaller than the one used by the `print()` function, and we are therefore able to see all columns side by side and without linebreak. Furthermore, it also shows the number of rows and columns in the DataFrame, which can be quite helpful in some cases.

Read

Refer to the link below for more details and examples on the `read_csv()` function of the pandas package:

https://pandas.pydata.org/docs/user_guide/io.html#io-read-csv-table

Read the following website for more information regarding the US Census data, including the explanation of the variable names and other useful information about the data:

http://www.cs.toronto.edu/~delve/data/adult/ adultDetail.html)

Chapter 2: Data Selection

Lesson Recording

Data Selection in pandas

Same as Python lists or NumPy arrays, we can access a pandas DataFrame by using the index operator `[]`. In this chapter, we will introduce three ways to subset rows, columns, or elements of a DataFrame.

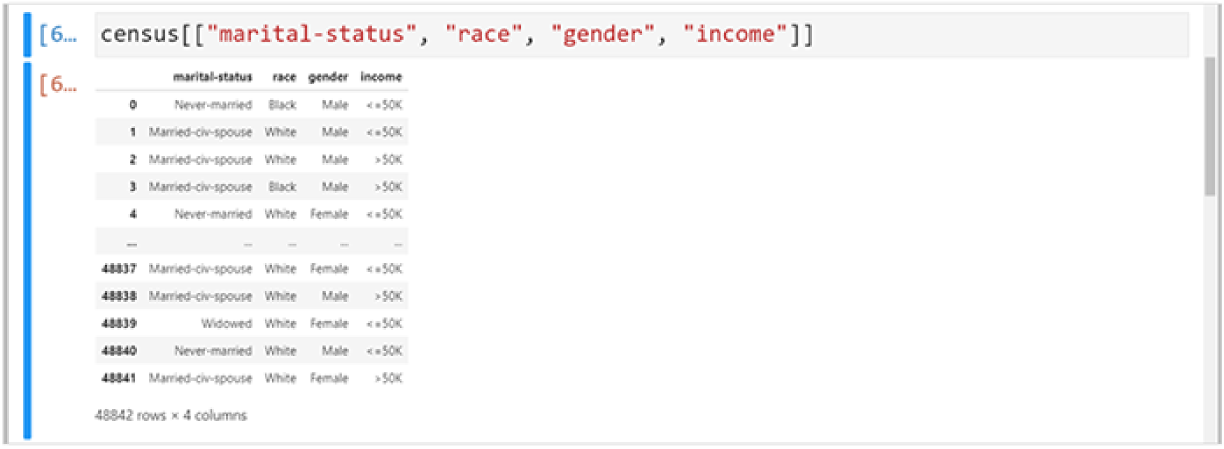
2.1 Selecting Columns by Variable Names

To select specific columns, which represent the variables of a dataset, we can create a list with the variable names (or labels) to be selected and then put it in the index operator.

```
DataFrame_name[["var_name1", "var_name2", …]]
```

Note that each variable name must be put within a pair of quotation marks since it is treated as a string in this case. If we simply want to access one column, we can omit the creation of the list and put the variable name as string inside the index operator directly.

Example (Cont'd): Now we would like to select the columns "`marital-status`", "`race`", "`gender`" and "`income`" from the US Census dataset.



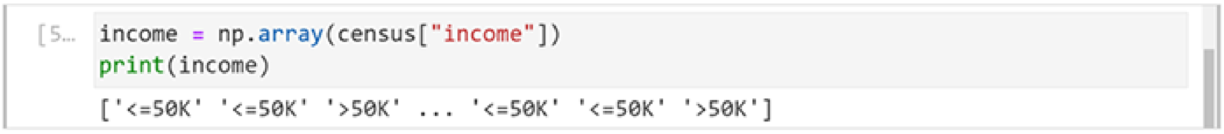
```
[6… census[["marital-status", "race", "gender", "income"]]
[6…
```

	marital-status	race	gender	income
0	Never-married	Black	Male	<=50K
1	Married-civ-spouse	White	Male	<=50K
2	Married-civ-spouse	White	Male	>50K
3	Married-civ-spouse	Black	Male	>50K
4	Never-married	White	Female	<=50K
...	...	...	...	...
48837	Married-civ-spouse	White	Female	<=50K
48838	Married-civ-spouse	White	Male	>50K
48839	Widowed	White	Female	<=50K
48840	Never-married	White	Male	<=50K
48841	Married-civ-spouse	White	Female	>50K

48842 rows × 4 columns

Figure 4.4 Select Columns from a DataFrame by Variable Names

Suppose we would like to select the target variable "`income`" alone and save the column as a NumPy array for further calculation.



```
[5… income = np.array(census["income"])
    print(income)
    ['<=50K' '<=50K' '>50K' ... '<=50K' '<=50K' '>50K']
```

Figure 4.5 Select a Single Column from a DataFrame and Save it as NumPy Array

We can see that no list will be needed within the index operator if only a single column is selected. After selecting the column, the resulting subset of the pandas DataFrame will then be converted to an NumPy array by the `np.array()` function.

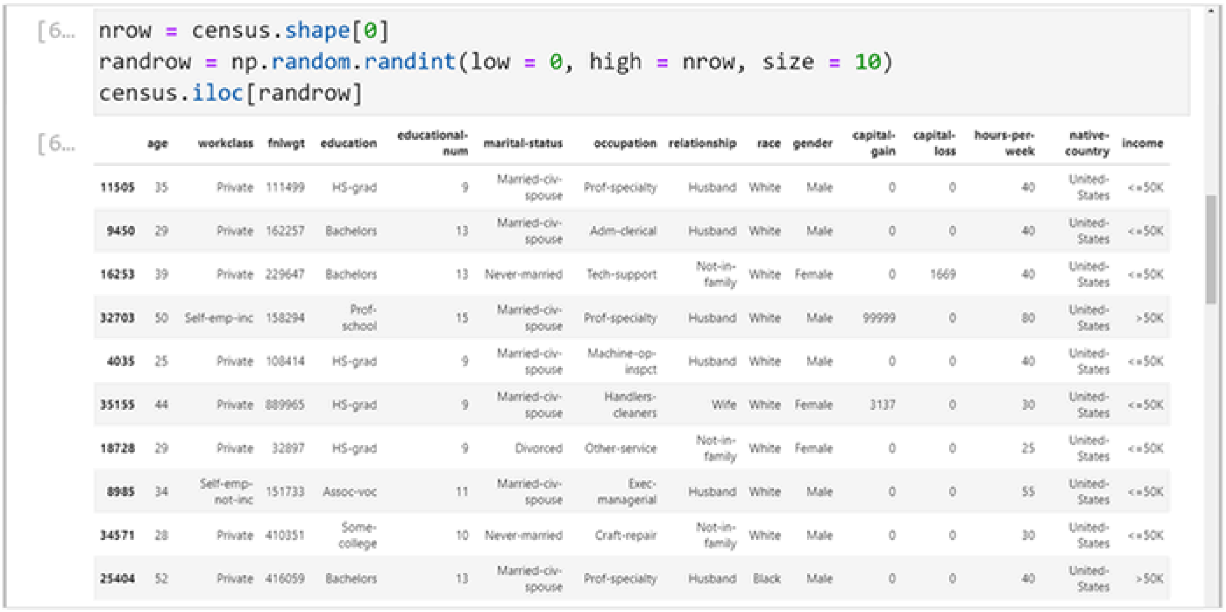
2.2 Selecting Rows by Positions and Indices

Accessing rows requires different techniques than accessing columns. While we can use the labels of the columns, or variable names, to select the columns we want, there are usually no natural "observation names" that we can refer to when selecting rows from a DataFrame. However, in Figure 4.1, we can see that a row index is provided at the beginning of every row by pandas. It starts with 0 and ends with the number of rows in the DataFrame minus one. As a result, rows can be queried by the numeric index position, starting at 0, using the DataFrame attribute `iloc`.

```
DataFrame_name.iloc[start:end]
```

The indices in the index operator do not need to be consecutive integers. It can be any integers within the range 0 and number of rows in the DataFrame – 1. But these integers must be put in a list first if there are more than one of them. If we want to select a single row instead, we can simply put one index in the index operator.

Example (Cont'd): Sometimes we may need to inspect a dataset after running some programs to adjust it for further analytics purposes. We have been introduced to the `.head()` method to print out the first five rows of the dataset. We can also randomly picked a few rows from the DataFrame for our inspection. Here, we need NumPy to draw random indices to select the rows for us.



```
[6...  nrow = census.shape[0]
       randrow = np.random.randint(low = 0, high = nrow, size = 10)
       census.iloc[randrow]
```

[6...

	age	workclass	fnlwgt	education	educational-num	marital-status	occupation	relationship	race	gender	capital-gain	capital-loss	hours-per-week	native-country	income
11505	35	Private	111499	HS-grad	9	Married-civ-spouse	Prof-specialty	Husband	White	Male	0	0	40	United-States	<=50K
9450	29	Private	162257	Bachelors	13	Married-civ-spouse	Adm-clerical	Husband	White	Male	0	0	40	United-States	<=50K
16253	39	Private	229647	Bachelors	13	Never-married	Tech-support	Not-in-family	White	Female	0	1669	40	United-States	<=50K
32703	50	Self-emp-inc	158294	Prof-school	15	Married-civ-spouse	Prof-specialty	Husband	White	Male	99999	0	80	United-States	>50K
4035	25	Private	108414	HS-grad	9	Married-civ-spouse	Machine-op-inspct	Husband	White	Male	0	0	40	United-States	<=50K
35155	44	Private	889965	HS-grad	9	Married-civ-spouse	Handlers-cleaners	Wife	White	Female	3137	0	30	United-States	<=50K
18728	29	Private	32897	HS-grad	9	Divorced	Other-service	Not-in-family	White	Female	0	0	25	United-States	<=50K
8985	34	Self-emp-not-inc	151733	Assoc-voc	11	Married-civ-spouse	Exec-managerial	Husband	White	Male	0	0	55	United-States	<=50K
34571	28	Private	410351	Some-college	10	Never-married	Craft-repair	Not-in-family	White	Male	0	0	30	United-States	<=50K
25404	52	Private	416059	Bachelors	13	Married-civ-spouse	Prof-specialty	Husband	Black	Male	0	0	40	United-States	>50K

Figure 4.6 Select Rows from a DataFrame by Random Row Indices

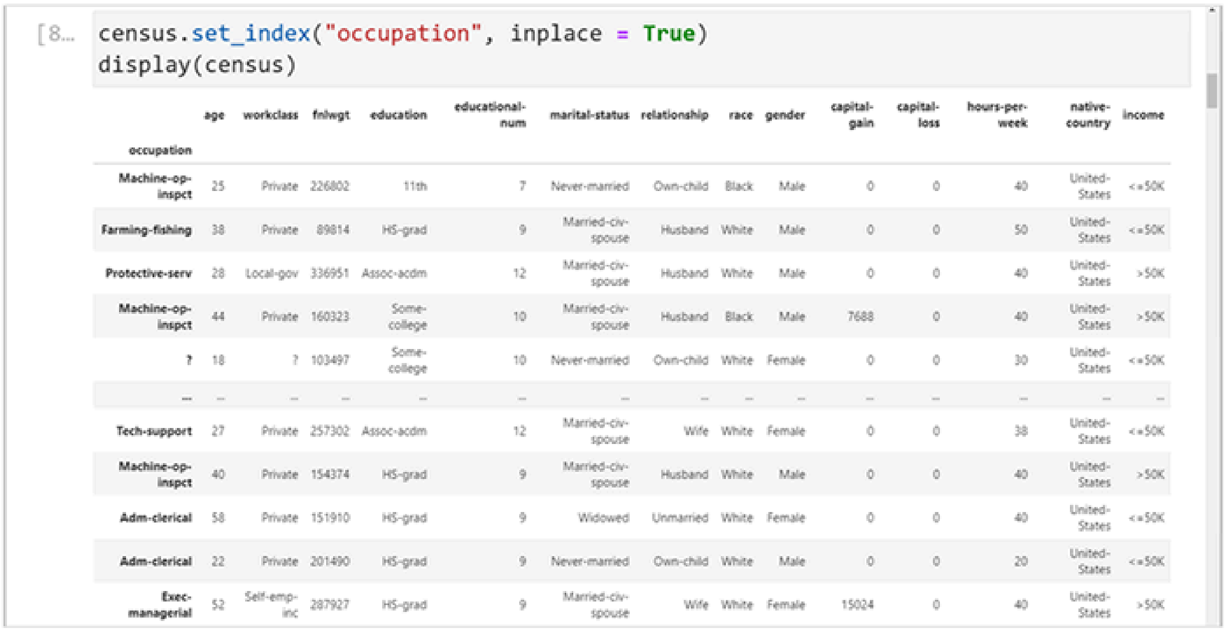
In the first line of our program, we determine the total number of rows in the dataset `census`. Note that the method `.shape()` is also applicable to pandas DataFrames and it returns a tuple `(Total Row Number, Total Column Number)` to us. As a result, we can refer to the first element of the tuple as the total number of rows in `census`. In the second line, we draw a total of 10 random integers from the interval 0 and `nrow`, the number of rows in `census`. Note that `nrow` as the upper boundary is not included in the drawing process at all. The integers drawn by the `random.randint()` function will then be assigned to the object named `randrow`. And these will then be used as the list of rows that we select from census.

Before we can select rows based on their index labels, we need to create the index labels first by the method `.set_index()`.

```
DataFrame_name.set_index(key, inplace = True)
```

The parameter `key` can be either a single variable name (column label), a single array of the same length as the calling DataFrame, or a list containing an arbitrary combination of variable names and arrays. The argument `inplace` controls whether the DataFrame should be modified in place or a new DataFrame should be created. If it is `True`, the changes will take place in the original DataFrame.

Example (Cont'd): Suppose we would like to group the US Census data by the occupation of the observations.



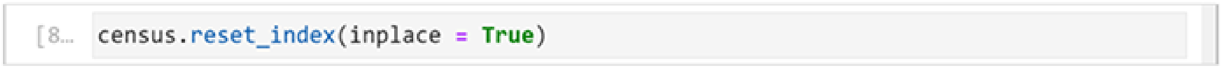
```
[8... census.set_index("occupation", inplace = True)
      display(census)
```

	age	workclass	fnlwgt	education	educational-num	marital-status	relationship	race	gender	capital-gain	capital-loss	hours-per-week	native-country	income
occupation														
Machine-op-inspct	25	Private	226802	11th	7	Never-married	Own-child	Black	Male	0	0	40	United-States	<=50K
Farming-fishing	38	Private	89814	HS-grad	9	Married-civ-spouse	Husband	White	Male	0	0	50	United-States	<=50K
Protective-serv	28	Local-gov	336951	Assoc-acdm	12	Married-civ-spouse	Husband	White	Male	0	0	40	United-States	>50K
Machine-op-inspct	44	Private	160323	Some-college	10	Married-civ-spouse	Husband	Black	Male	7688	0	40	United-States	>50K
?	18	?	103497	Some-college	10	Never-married	Own-child	White	Female	0	0	30	United-States	<=50K
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
Tech-support	27	Private	257302	Assoc-acdm	12	Married-civ-spouse	Wife	White	Female	0	0	38	United-States	<=50K
Machine-op-inspct	40	Private	154374	HS-grad	9	Married-civ-spouse	Husband	White	Male	0	0	40	United-States	>50K
Adm-clerical	58	Private	151910	HS-grad	9	Widowed	Unmarried	White	Female	0	0	40	United-States	<=50K
Adm-clerical	22	Private	201490	HS-grad	9	Never-married	Own-child	White	Male	0	0	20	United-States	<=50K
Exec-managerial	52	Self-emp-inc	287927	HS-grad	9	Married-civ-spouse	Wife	White	Female	15024	0	40	United-States	>50K

Figure 4.7 Setting the Values of a Column as Row Index

If we use one of the columns as the row index of our DataFrame, that column will not be a regular part of the dataset anymore. As you can see from Figure 4.7, the variable `occupation` has disappeared from the dataset.

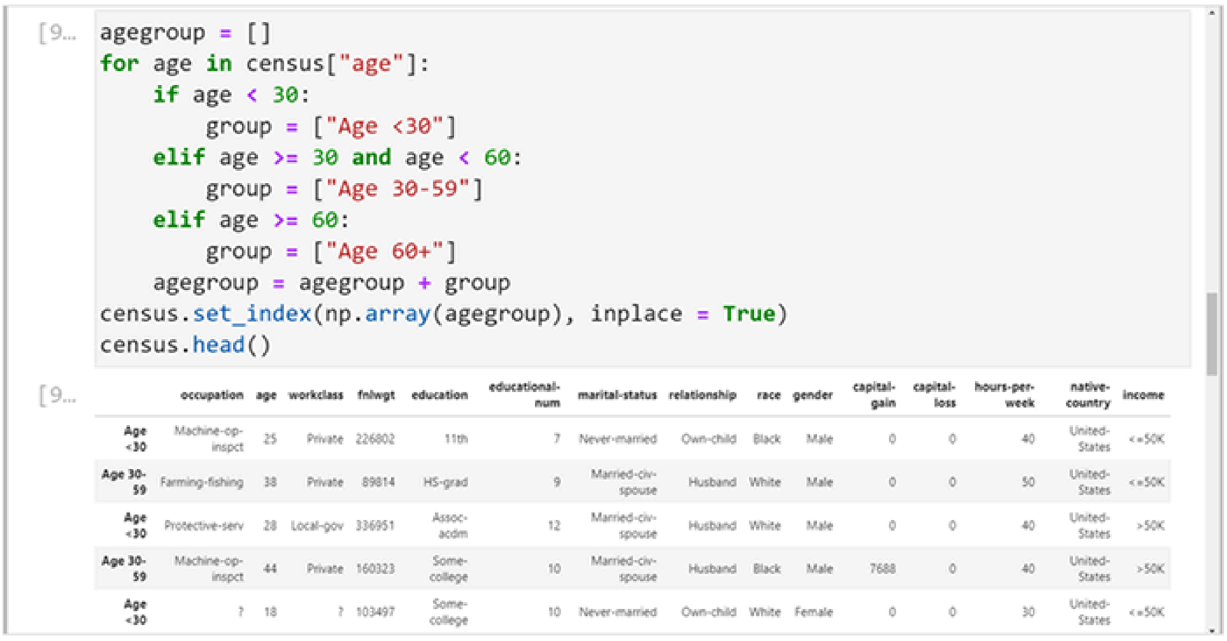
Suppose we would like to change our row index from the occupation to the age group of each observation. In the first step, we need to remove the occupation as our row index by the `.reset_index()` method. It reverses the effect of the `.set_index()` method and removes the current row index and converts it back to a column in the DataFrame.



```
[8... census.reset_index(inplace = True)
```

Figure 4.8 Resetting Row Index of a DataFrame

Then we recode the age in each row to our target grouping and store it in a new array. For observations less than 30 years, the group will be labelled as "Age <30", whereas the label of those between 30 and 59 years old is "Age 30-59" and the rest receives the label "Age 60+". Subsequently, we assign this array as the new row index.



```
[9...
agegroup = []
for age in census["age"]:
    if age < 30:
        group = ["Age <30"]
    elif age >= 30 and age < 60:
        group = ["Age 30-59"]
    elif age >= 60:
        group = ["Age 60+"]
    agegroup = agegroup + group
census.set_index(np.array(agegroup), inplace = True)
census.head()
```

[9...

	occupation	age	workclass	fnlwgt	education	educational-num	marital-status	relationship	race	gender	capital-gain	capital-loss	hours-per-week	native-country	income
Age <30	Machine-op-inspct	25	Private	226802	11th	7	Never-married	Own-child	Black	Male	0	0	40	United-States	<=50K
Age 30-59	Farming-fishing	38	Private	89814	HS-grad	9	Married-civ-spouse	Husband	White	Male	0	0	50	United-States	<=50K
Age <30	Protective-serv	28	Local-gov	336951	Assoc-acdm	12	Married-civ-spouse	Husband	White	Male	0	0	40	United-States	>50K
Age 30-59	Machine-op-inspct	44	Private	160323	Some-college	10	Married-civ-spouse	Husband	Black	Male	7688	0	40	United-States	>50K
Age <30	?	18	?	103497	Some-college	10	Never-married	Own-child	White	Female	0	0	30	United-States	<=50K

Figure 4.9 Setting a NumPy Array as Row Index

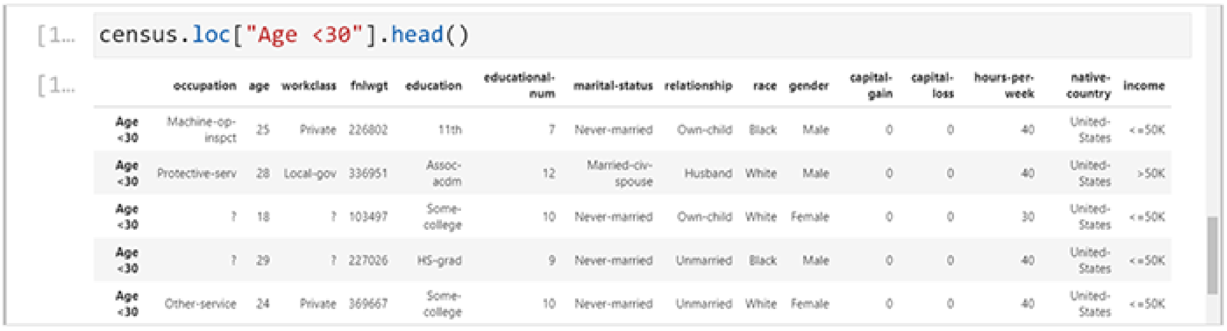
We first create a list named `agegroup` that contains the grouping of each observation's age. The list will then be converted into a NumPy array and used as the row index subsequently.

The DataFrame rows can be queried by the row index labels using the `.loc` attribute.

```
DataFrame_name.loc[["row_label1", "row_label2", …]]
```

We can see that selecting rows from a DataFrame by the `.loc` attribute works in a very similar fashion as the column selection. The row labels must be indicated as strings and put in a list if we want to select more than one of them. If we just want to select rows of a single label, we can put the label as a string in the `.loc` attribute directly.

Example (Cont'd): Suppose we would like to select all the observations that are 30 years of age or younger from the US Census data.



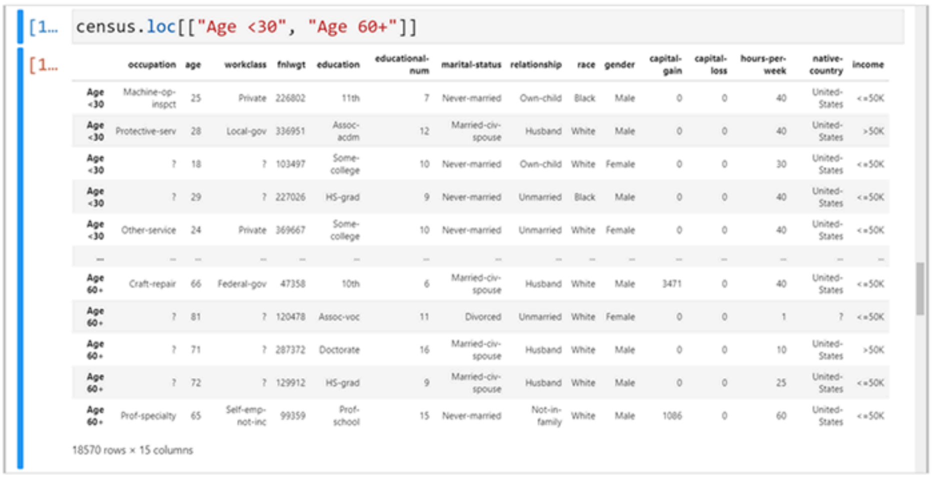
```
[1… census.loc["Age <30"].head()
```

[1…

	occupation	age	workclass	fnlwgt	education	educational-num	marital-status	relationship	race	gender	capital-gain	capital-loss	hours-per-week	native-country	income
Age <30	Machine-op-inspct	25	Private	226802	11th	7	Never-married	Own-child	Black	Male	0	0	40	United-States	<=50K
Age <30	Protective-serv	28	Local-gov	336951	Assoc-acdm	12	Married-civ-spouse	Husband	White	Male	0	0	40	United-States	>50K
Age <30	?	18	?	103497	Some-college	10	Never-married	Own-child	White	Female	0	0	30	United-States	<=50K
Age <30	?	29	?	227026	HS-grad	9	Never-married	Unmarried	Black	Male	0	0	40	United-States	<=50K
Age <30	Other-service	24	Private	369667	Some-college	10	Never-married	Unmarried	White	Female	0	0	40	United-States	<=50K

Figure 4.10 Selecting Rows by Single Row Label

And if we want to select the youngest and oldest age groups from `census`, we will need to put the row labels in a list.



```
[1… census.loc[["Age <30", "Age 60+"]]
```

[1…

	occupation	age	workclass	fnlwgt	education	educational-num	marital-status	relationship	race	gender	capital-gain	capital-loss	hours-per-week	native-country	income
Age <30	Machine-op-inspct	25	Private	226802	11th	7	Never-married	Own-child	Black	Male	0	0	40	United-States	<=50K
Age <30	Protective-serv	28	Local-gov	336951	Assoc-acdm	12	Married-civ-spouse	Husband	White	Male	0	0	40	United-States	>50K
Age <30	?	18	?	103497	Some-college	10	Never-married	Own-child	White	Female	0	0	30	United-States	<=50K
Age <30	?	29	?	227026	HS-grad	9	Never-married	Unmarried	Black	Male	0	0	40	United-States	<=50K
Age <30	Other-service	24	Private	369667	Some-college	10	Never-married	Unmarried	White	Female	0	0	40	United-States	<=50K
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
Age 60+	Craft-repair	66	Federal-gov	47358	10th	6	Married-civ-spouse	Husband	White	Male	3471	0	40	United-States	<=50K
Age 60+	?	81	?	120478	Assoc-voc	11	Divorced	Unmarried	White	Female	0	0	1	?	<=50K
Age 60+	?	71	?	287372	Doctorate	16	Married-civ-spouse	Husband	White	Male	0	0	10	United-States	>50K
Age 60+	?	72	?	129912	HS-grad	9	Married-civ-spouse	Husband	White	Male	0	0	25	United-States	<=50K
Age 60+	Prof-specialty	65	Self-emp-not-inc	99359	Prof-school	15	Never-married	Not-in-family	White	Male	1086	0	60	United-States	<=50K

18570 rows × 15 columns

Figure 4.11 Selecting Rows by Multiple Row Labels

Read

Refer to the links below for more details and examples on the attributes `.set_index()` and `.reset_index()` of the pandas package:

https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.set_index.html

https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.reset_index.html

Refer to the links below for more details and examples on the methods `.loc()` and `.iloc()` of the pandas package:

https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.loc.html

https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.iloc.html

2.3 Selecting Cells by Positions and Indices

To select elements in the DataFrame, we can specify both column and row labels in the `.loc` attribute, or the positions in the `.iloc` attribute, or a combination of both.

Below is a syntax that uses only the row and column indices for the cell selection.

```
DataFrame_name.iloc[row_start:row_end, col_start:col_end]
```

We can also select cells by referring to the corresponding row and column labels.

```
DataFrame_name.loc[["row_labels"], ["col_labels"]]
```

If we want to select the rows by index but the columns by labels, we can use the index operator and `.iloc` attribute together.

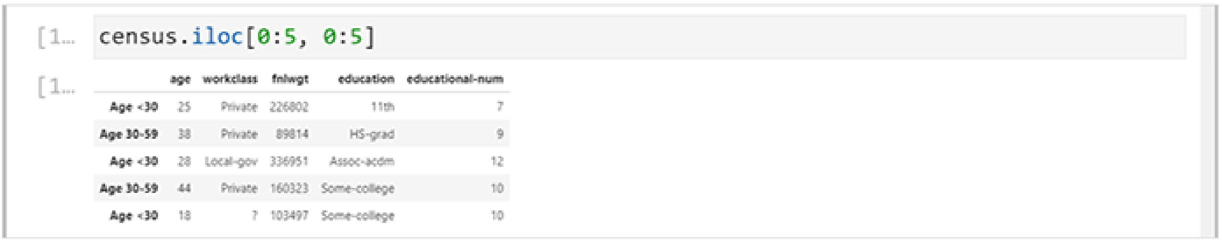
```
DataFrame_name[["col_labels"]].iloc[row_start:row_end]
```

But if we want to select the columns by index but the rows by labels, we need to use both the `.loc` and `.iloc` attributes.

```
DataFrame_name.loc[["row_labels"]].iloc[:,
col_start:col_end]
```

While putting the row labels in the `.loc` attribute, we need to be aware that the `.iloc` attribute requires both the row and column indices. Since we do not intend to select the rows by index, we can use the open-end index `0:` or simply `:` here.

Example (Cont'd): Suppose we would like to select the first five observations of the first five variables.

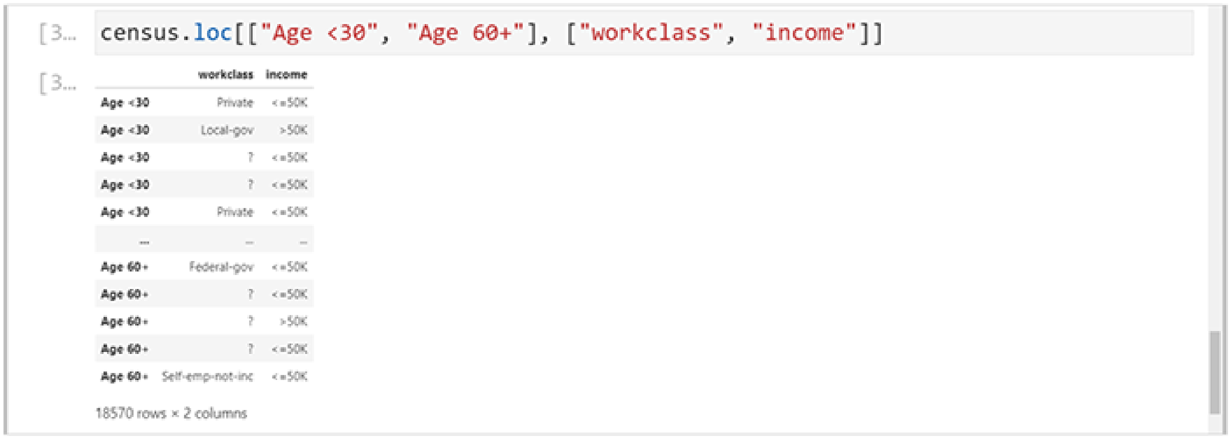


```
[1… census.iloc[0:5, 0:5]
```

```
[1…            age  workclass  fnlwgt     education  educational-num
    Age <30     25    Private  226802          11th                7
    Age 30-59   38    Private   89814       HS-grad                9
    Age <30     28  Local-gov  336951    Assoc-acdm               12
    Age 30-59   44    Private  160323  Some-college               10
    Age <30     18          ?  103497  Some-college               10
```

Figure 4.12 Selecting Cells Using Indices

Such a selection only makes sense when we exactly know the positions of the variables and observations. More common is the selection of cells based on labels. Suppose we would like to select the observed values in `workclass` and `income` for all observations younger than 30 and older than 60 years old.

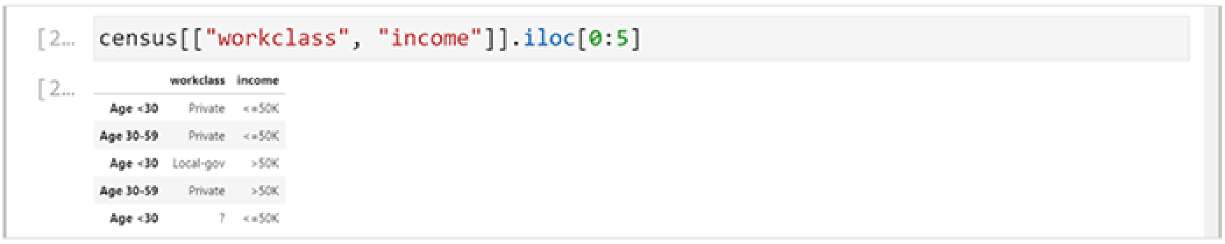


```
[3... census.loc[["Age <30", "Age 60+"], ["workclass", "income"]]
```

```
[3...
                  workclass   income
Age <30             Private    <=50K
Age <30           Local-gov     >50K
Age <30                   ?    <=50K
Age <30                   ?    <=50K
Age <30             Private    <=50K
    ...                 ...      ...
Age 60+         Federal-gov    <=50K
Age 60+                   ?    <=50K
Age 60+                   ?     >50K
Age 60+                   ?    <=50K
Age 60+    Self-emp-not-inc    <=50K

18570 rows × 2 columns
```

Figure 4.13 Selecting Cells Using Row and Column Labels

Nevertheless, this selection method only works if the row labels are set and we can refer to them in the `.loc` attribute. While column labels usually correspond to the variable names, row labels are not necessarily used as row index in most of the datasets. The row positions are therefore more useful in selecting cells in a DataFrame. Suppose we would now like to select the observed values of `workclass` and `income` from the first 5 rows.

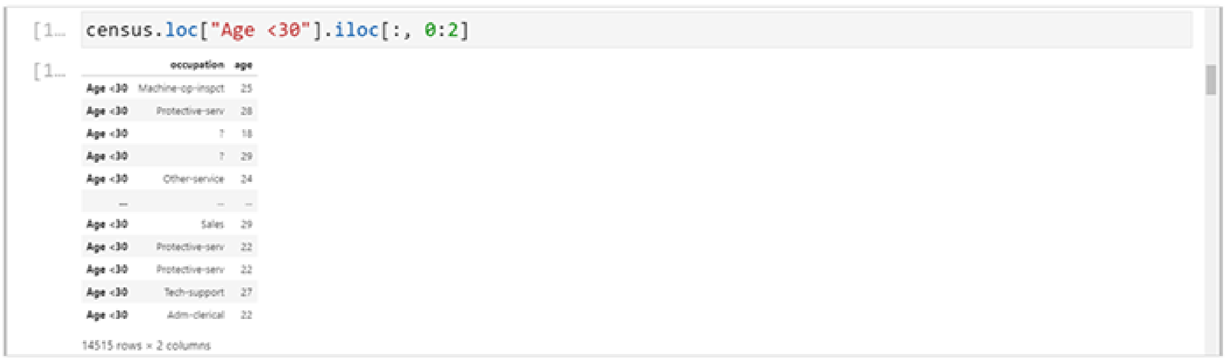


```
[2... census[["workclass", "income"]].iloc[0:5]
```

```
[2...
             workclass   income
    Age <30    Private    <=50K
  Age 30-59    Private    <=50K
    Age <30  Local-gov     >50K
  Age 30-59    Private     >50K
    Age <30          ?    <=50K
```

Figure 4.14 Selecting Cells by Row Indices and Column Labels

If we want to select the observed values of the first two columns from the observations younger than 30 years old, we will have to use the `.iloc` and `.loc` attributes at the same time.



```
[1… census.loc["Age <30"].iloc[:, 0:2]
```

```
[1…
            occupation          age
Age <30     Machine-op-inspct   25
Age <30     Protective-serv     28
Age <30     ?                   18
Age <30     ?                   29
Age <30     Other-service       24
…           …                   …
Age <30     Sales               29
Age <30     Protective-serv     22
Age <30     Protective-serv     22
Age <30     Tech-support        27
Age <30     Adm-clerical        22
14515 rows × 2 columns
```

Figure 4.15 Selecting Cells by Row Labels and Column Indices

2.4 Selecting Cells by Boolean Masking

In Chapter 2.2 of Study Unit 3, we learned how to use Boolean mask to subset a NumPy array. Here, we will apply the same technique to select cells from a DataFrame. A Boolean mask is an array where each of the values is either `True` or `False`. The Boolean mask array is overlaid on top of the data structure that we're querying. And any element aligned with a `True` value will be selected, and any element aligned with a `False` value will not.

```
DataFrame_name[Condition]
```

We can also create more complex queries by using bitwise logical operators to chain several conditions together.

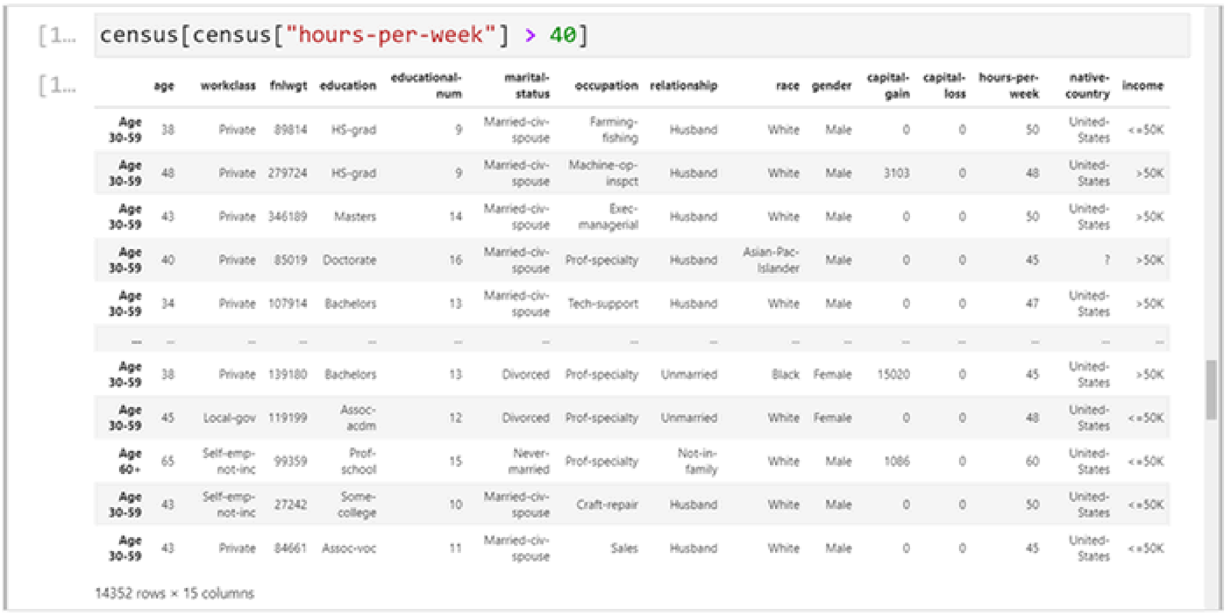
```
DataFrame_name[(Condition1) &/| (Condition2) &/| …]
```

The bitwise logical operators are similar to the logical operators. Instead of writing `and`/`or`, we use `&` (bitwise `and`), `|` (bitwise `or`), or `~` (bitwise `not`) to combine our conditions in the DataFrame queries. We can also add the bitwise `not` operator to the above syntax if we want to negate any condition.

We need the bitwise logical operators here because we are actually creating a Boolean mask for each condition within the index operator `[]`. If there are two conditions, two Boolean masks will be compared elementwise by the bitwise operator. The result of this comparison is in turn a Boolean mask as well.

Remember that each Boolean mask / condition needs to be encased in parentheses because of the order of operations.

Example (Cont'd): Suppose we would like to select those observations that work more than 40 hours per week.



```
[1...  census[census["hours-per-week"] > 40]
```

[1...

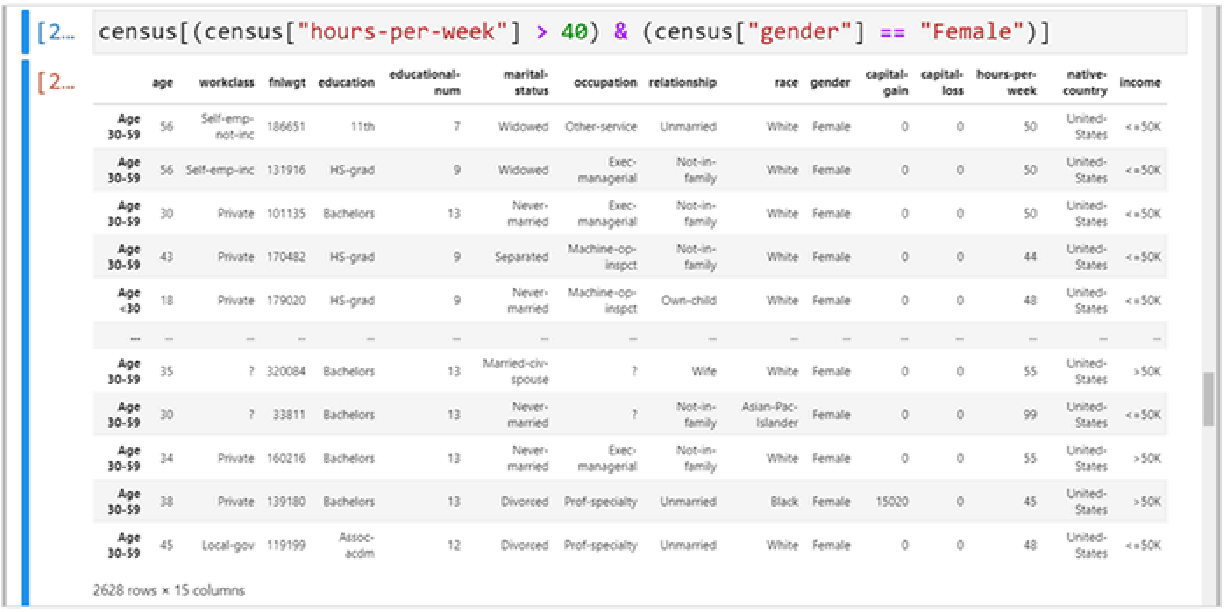
	age	workclass	fnlwgt	education	educational-num	marital-status	occupation	relationship	race	gender	capital-gain	capital-loss	hours-per-week	native-country	income
Age 30-59	38	Private	89814	HS-grad	9	Married-civ-spouse	Farming-fishing	Husband	White	Male	0	0	50	United-States	<=50K
Age 30-59	48	Private	279724	HS-grad	9	Married-civ-spouse	Machine-op-inspct	Husband	White	Male	3103	0	48	United-States	>50K
Age 30-59	43	Private	346189	Masters	14	Married-civ-spouse	Exec-managerial	Husband	White	Male	0	0	50	United-States	>50K
Age 30-59	40	Private	85019	Doctorate	16	Married-civ-spouse	Prof-specialty	Husband	Asian-Pac-Islander	Male	0	0	45	?	>50K
Age 30-59	34	Private	107914	Bachelors	13	Married-civ-spouse	Tech-support	Husband	White	Male	0	0	47	United-States	>50K
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
Age 30-59	38	Private	139180	Bachelors	13	Divorced	Prof-specialty	Unmarried	Black	Female	15020	0	45	United-States	>50K
Age 30-59	45	Local-gov	119199	Assoc-acdm	12	Divorced	Prof-specialty	Unmarried	White	Female	0	0	48	United-States	<=50K
Age 60+	65	Self-emp-not-inc	99359	Prof-school	15	Never-married	Prof-specialty	Not-in-family	White	Male	1086	0	60	United-States	<=50K
Age 30-59	43	Self-emp-not-inc	27242	Some-college	10	Married-civ-spouse	Craft-repair	Husband	White	Male	0	0	50	United-States	<=50K
Age 30-59	43	Private	84661	Assoc-voc	11	Married-civ-spouse	Sales	Husband	White	Male	0	0	45	United-States	<=50K

14352 rows × 15 columns

Figure 4.16 Selecting Cells by a Boolean Mask

The above syntax is a combination of two instructions. First, we create a Boolean mask to select only those observations where `census["hours-per-week"] > 40` is `True`. Recall that `census["hours-per-week"]` is actually a syntax to select a specific column from the DataFrame. So, the condition here is to tell Python to select the column named `hours-per-week` first and then assign `True` to those cases where the observed value is larger than 40. Those row indices where the Boolean mask is `True` will then be selected from the DataFrame `census` by the index operator `[]`.

In the next query, we would like to select female respondents from the DataFrame that work more than 40 hours per week.



```
[2… census[(census["hours-per-week"] > 40) & (census["gender"] == "Female")]
```

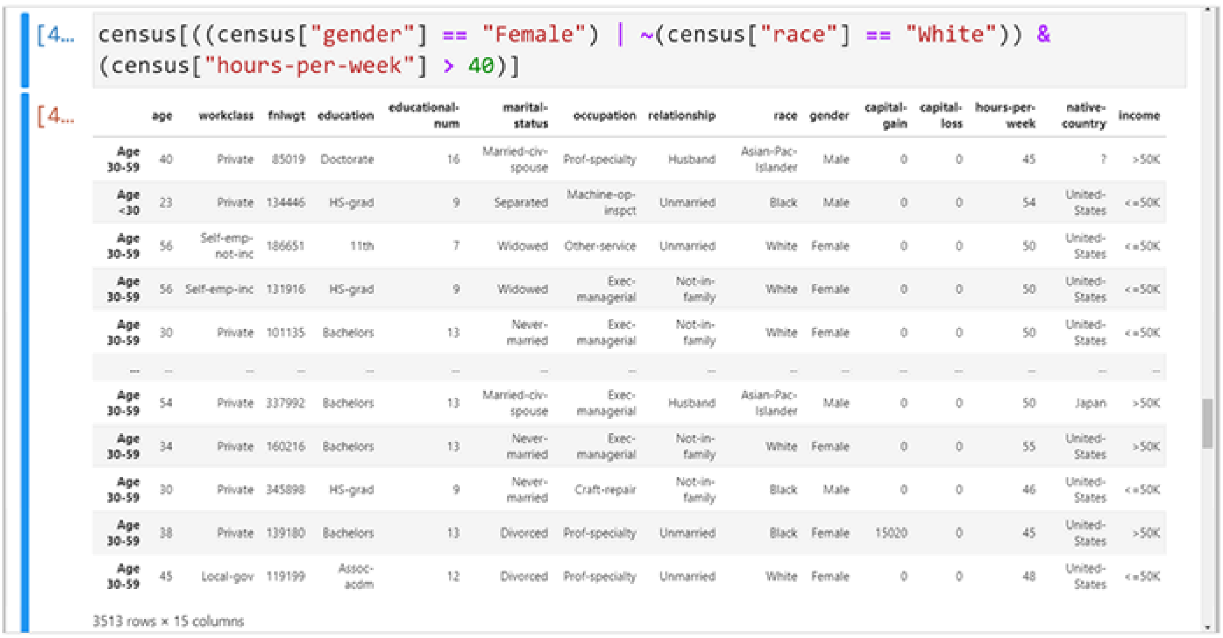
[2…

	age	workclass	fnlwgt	education	educational-num	marital-status	occupation	relationship	race	gender	capital-gain	capital-loss	hours-per-week	native-country	income
Age 30-59	56	Self-emp-not-inc	186651	11th	7	Widowed	Other-service	Unmarried	White	Female	0	0	50	United-States	<=50K
Age 30-59	56	Self-emp-inc	131916	HS-grad	9	Widowed	Exec-managerial	Not-in-family	White	Female	0	0	50	United-States	<=50K
Age 30-59	30	Private	101135	Bachelors	13	Never-married	Exec-managerial	Not-in-family	White	Female	0	0	50	United-States	<=50K
Age 30-59	43	Private	170482	HS-grad	9	Separated	Machine-op-inspct	Not-in-family	White	Female	0	0	44	United-States	<=50K
Age <30	18	Private	179020	HS-grad	9	Never-married	Machine-op-inspct	Own-child	White	Female	0	0	48	United-States	<=50K
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
Age 30-59	35	?	320084	Bachelors	13	Married-civ-spouse	?	Wife	White	Female	0	0	55	United-States	>50K
Age 30-59	30	?	33811	Bachelors	13	Never-married	?	Not-in-family	Asian-Pac-Islander	Female	0	0	99	United-States	<=50K
Age 30-59	34	Private	160216	Bachelors	13	Never-married	Exec-managerial	Not-in-family	White	Female	0	0	55	United-States	>50K
Age 30-59	38	Private	139180	Bachelors	13	Divorced	Prof-specialty	Unmarried	Black	Female	15020	0	45	United-States	>50K
Age 30-59	45	Local-gov	119199	Assoc-acdm	12	Divorced	Prof-specialty	Unmarried	White	Female	0	0	48	United-States	<=50K

2628 rows × 15 columns

Figure 4.17 Selecting Cells by Chaining Two Boolean Masks

In the last query, we want to select female or non-white respondents who work more than 40 hours per week.



```
[4… census[((census["gender"] == "Female") | ~(census["race"] == "White")) &
    (census["hours-per-week"] > 40)]
```

[4…

	age	workclass	fnlwgt	education	educational-num	marital-status	occupation	relationship	race	gender	capital-gain	capital-loss	hours-per-week	native-country	income
Age 30-59	40	Private	85019	Doctorate	16	Married-civ-spouse	Prof-specialty	Husband	Asian-Pac-Islander	Male	0	0	45	?	>50K
Age <30	23	Private	134446	HS-grad	9	Separated	Machine-op-inspct	Unmarried	Black	Male	0	0	54	United-States	<=50K
Age 30-59	56	Self-emp-not-inc	186651	11th	7	Widowed	Other-service	Unmarried	White	Female	0	0	50	United-States	<=50K
Age 30-59	56	Self-emp-inc	131916	HS-grad	9	Widowed	Exec-managerial	Not-in-family	White	Female	0	0	50	United-States	<=50K
Age 30-59	30	Private	101135	Bachelors	13	Never-married	Exec-managerial	Not-in-family	White	Female	0	0	50	United-States	<=50K
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
Age 30-59	54	Private	337992	Bachelors	13	Married-civ-spouse	Exec-managerial	Husband	Asian-Pac-Islander	Male	0	0	50	Japan	>50K
Age 30-59	34	Private	160216	Bachelors	13	Never-married	Exec-managerial	Not-in-family	White	Female	0	0	55	United-States	>50K
Age 30-59	30	Private	345898	HS-grad	9	Never-married	Craft-repair	Not-in-family	Black	Male	0	0	46	United-States	<=50K
Age 30-59	38	Private	139180	Bachelors	13	Divorced	Prof-specialty	Unmarried	Black	Female	15020	0	45	United-States	>50K
Age 30-59	45	Local-gov	119199	Assoc-acdm	12	Divorced	Prof-specialty	Unmarried	White	Female	0	0	48	United-States	<=50K

3513 rows × 15 columns

Figure 4.18 Selecting Cells by Chaining Multiple Boolean Masks

In the first parentheses, we create a Boolean mask for observations where `gender` is equal to `"Female"`. In the second parentheses, the Boolean mask is created for observations where `race` is *not* `"White"`. These two masks are compared by the `|` (bitwise `or`) operator. The resulting Boolean mask will then be compared by a Boolean mask where the values in `hours-per-week` are larger than 40.

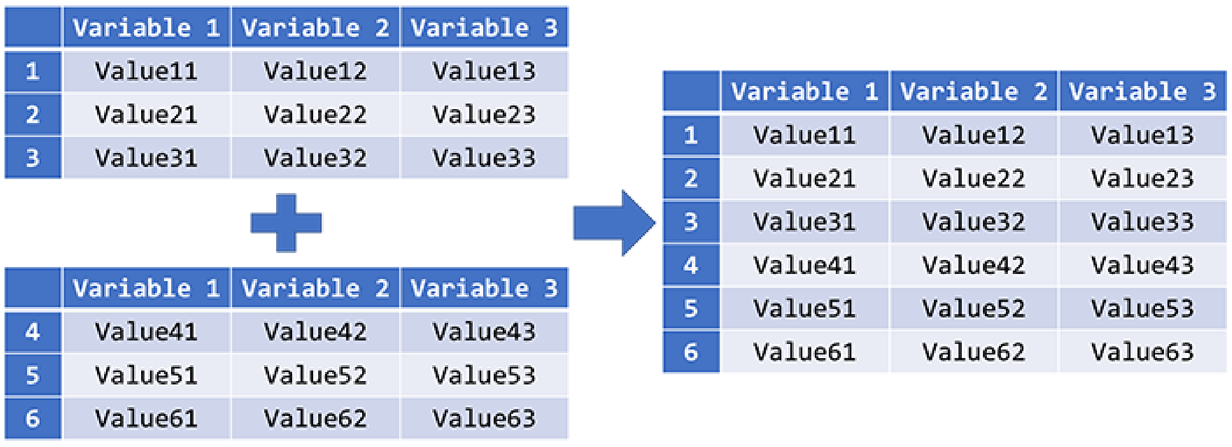
Chapter 3: Merge DataFrames

Lesson Recording

Merge DataFrames in pandas

3.1 Appending DataFrames by Rows

It often happens that multiple parties are actually collecting data for the same empirical study simultaneously. Eventually, their collected data must be merged together for analyses. Though the data could be collected at different locations or during different periods, they must consist of the same variables since the study is identical. Merging these datasets means to append their rows below each other to become one dataset.



	Variable 1	Variable 2	Variable 3
1	Value11	Value12	Value13
2	Value21	Value22	Value23
3	Value31	Value32	Value33

	Variable 1	Variable 2	Variable 3
4	Value41	Value42	Value43
5	Value51	Value52	Value53
6	Value61	Value62	Value63

	Variable 1	Variable 2	Variable 3
1	Value11	Value12	Value13
2	Value21	Value22	Value23
3	Value31	Value32	Value33
4	Value41	Value42	Value43
5	Value51	Value52	Value53
6	Value61	Value62	Value63

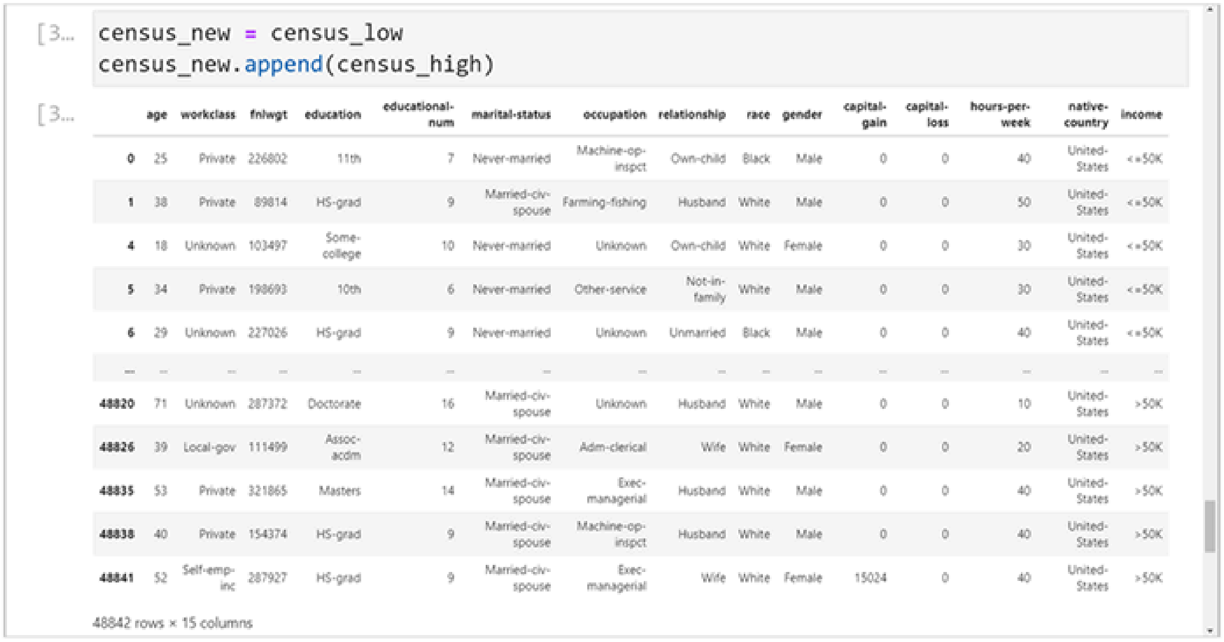
Figure 4.19 Concatenating Two Datasets with Different Rows but Identical Variables

In Python, we can use the `.append()` method to merge two DataFrames with identical variables into one.

```
DataFrame_name.append(other = [OtherDataFrames])
```

The parameter `other` is used for the specification of those DataFrames to be appended to `DataFrame_name` eventually. If we only have one DataFrame to be assigned to the parameter `other`, we simply put its name without quotation marks behind `other =`. In the case of specifying multiple DataFrames to the parameter `other`, we need to put their names in a list.

Example (Cont'd): In order to study the different income groups in the US census data more efficiently, the data analysts have decided to split the dataset into two. Observatons with income "<50K" will be saved in a new dataset named "census_low" and those with income ">50K" are now saved in "census_high". Now, after the datasets have been cleaned and studied separately, both datasets should be merged again for some joint analyses.



```
[3... census_new = census_low
      census_new.append(census_high)
```

[3...

	age	workclass	fnlwgt	education	educational-num	marital-status	occupation	relationship	race	gender	capital-gain	capital-loss	hours-per-week	native-country	income
0	25	Private	226802	11th	7	Never-married	Machine-op-inspct	Own-child	Black	Male	0	0	40	United-States	<=50K
1	38	Private	89814	HS-grad	9	Married-civ-spouse	Farming-fishing	Husband	White	Male	0	0	50	United-States	<=50K
4	18	Unknown	103497	Some-college	10	Never-married	Unknown	Own-child	White	Female	0	0	30	United-States	<=50K
5	34	Private	198693	10th	6	Never-married	Other-service	Not-in-family	White	Male	0	0	30	United-States	<=50K
6	29	Unknown	227026	HS-grad	9	Never-married	Unknown	Unmarried	Black	Male	0	0	40	United-States	<=50K
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
48820	71	Unknown	287372	Doctorate	16	Married-civ-spouse	Unknown	Husband	White	Male	0	0	10	United-States	>50K
48826	39	Local-gov	111499	Assoc-acdm	12	Married-civ-spouse	Adm-clerical	Wife	White	Female	0	0	20	United-States	>50K
48835	53	Private	321865	Masters	14	Married-civ-spouse	Exec-managerial	Husband	White	Male	0	0	40	United-States	>50K
48838	40	Private	154374	HS-grad	9	Married-civ-spouse	Machine-op-inspct	Husband	White	Male	0	0	40	United-States	>50K
48841	52	Self-emp-inc	287927	HS-grad	9	Married-civ-spouse	Exec-managerial	Wife	White	Female	15024	0	40	United-States	>50K

48842 rows × 15 columns

Figure 4.20 Appending Two DataFrames

Note that we can also apply the `.append()` method on `census_low` directly. However, we must be very sure that we no longer need `census_low` with its original data since there is no way to retrieve its original content after the appending process, unless we can import the original dataset from an external source again. Furthermore, if we appended `census_high` to `census_low` directly and re-ran the same code out of whatever reasons, `census_low` would eventually contain the observations of `census_high` double. Therefore, in Figure 4.20, we first copy `census_low` to a new DataFrame named `census_new`. And the `.append()` method is only applied on `census_new`. Re-running the same code would not create mess in any of the involved DataFrames at all.

Logically, the observations in the merged DataFrame do not follow the same order as `census` since it does not play any role in this appending process at all.

Read

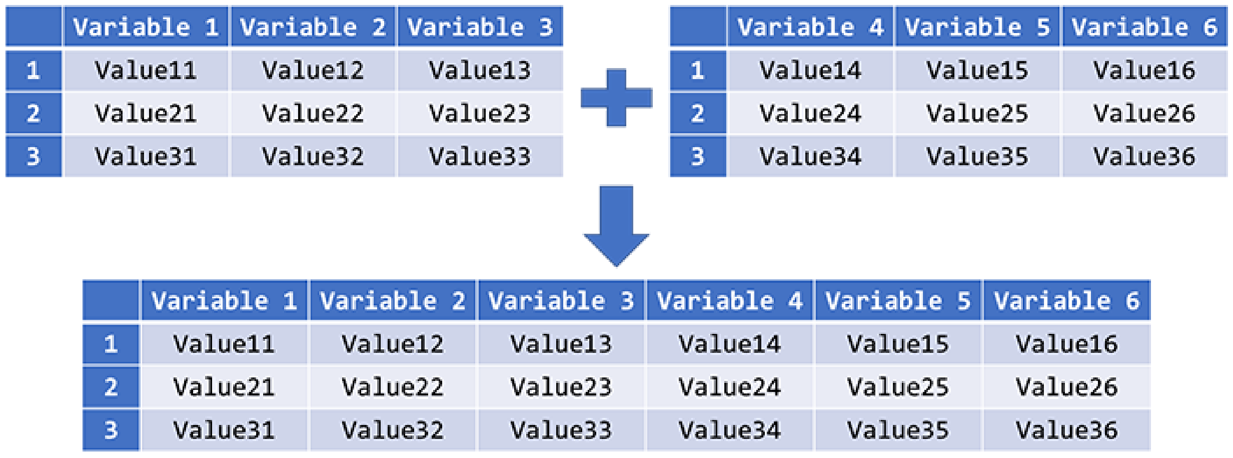
Refer to the link below for more details and examples on the `.append()` method of the pandas package:

https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.append.html

3.2 Merging DataFrames with Different Shapes

The `.append()` method introduced in Chapter 3.1 is applicable for merging two DataFrames with the same variables by rows. Nevertheless, there are other scenarios when merging multiple datasets in general.

Another rather uncomplicated scenario is that different variables are found across multiple DataFrames. But they contain the same observations.



	Variable 1	Variable 2	Variable 3
1	Value11	Value12	Value13
2	Value21	Value22	Value23
3	Value31	Value32	Value33

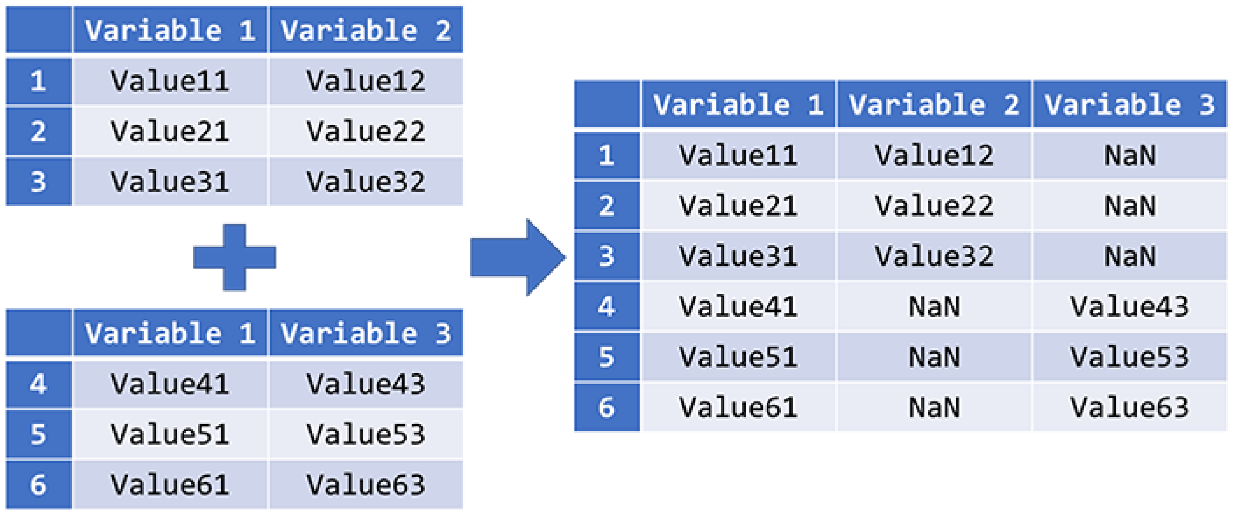
	Variable 4	Variable 5	Variable 6
1	Value14	Value15	Value16
2	Value24	Value25	Value26
3	Value34	Value35	Value36

	Variable 1	Variable 2	Variable 3	Variable 4	Variable 5	Variable 6
1	Value11	Value12	Value13	Value14	Value15	Value16
2	Value21	Value22	Value23	Value24	Value25	Value26
3	Value31	Value32	Value33	Value34	Value35	Value36

Figure 4.21 Concatenating Two Datasets with Different Columns but Identical Observations

For merging DataFrames by columns, they need to have identical keys, which are usually the row labels of the DataFrames. As Figure 4.21 illustrates, Python can use the row labels of both DataFrames to match identical observations and append their values of all the available variables in both DataFrames in the same row.

A more complicated scenario is that we have multiple DataFrames with some common variables but completely different observations.

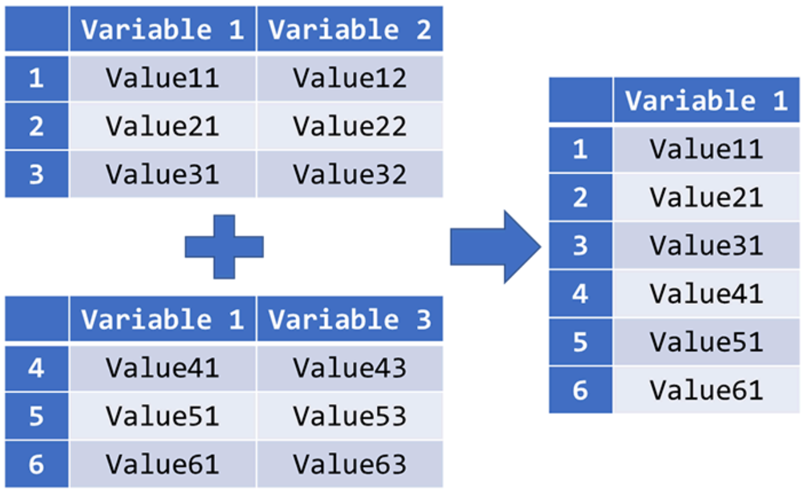


	Variable 1	Variable 2
1	Value11	Value12
2	Value21	Value22
3	Value31	Value32

	Variable 1	Variable 3
4	Value41	Value43
5	Value51	Value53
6	Value61	Value63

	Variable 1	Variable 2	Variable 3
1	Value11	Value12	NaN
2	Value21	Value22	NaN
3	Value31	Value32	NaN
4	Value41	NaN	Value43
5	Value51	NaN	Value53
6	Value61	NaN	Value63

Figure 4.22 Outer Join Two Datasets with Some Common Variables



	Variable 1	Variable 2
1	Value11	Value12
2	Value21	Value22
3	Value31	Value32

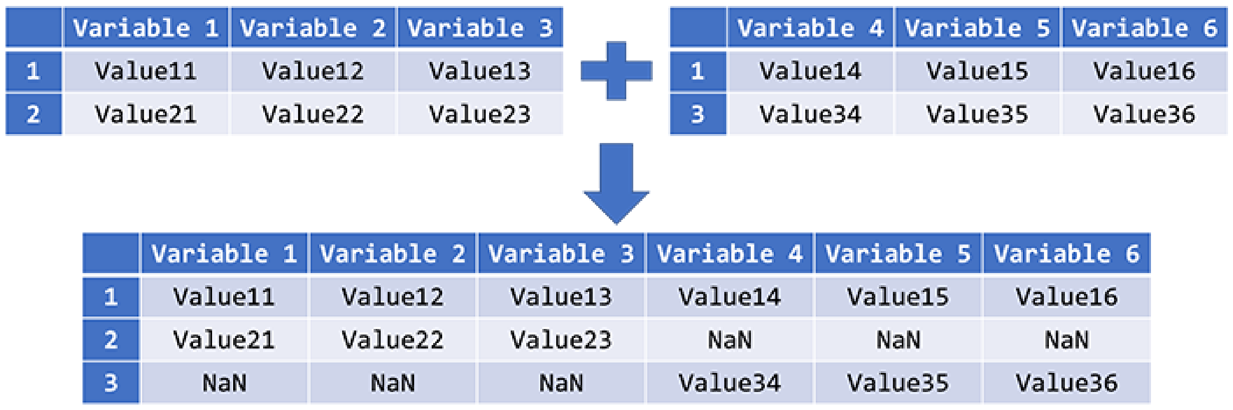
	Variable 1	Variable 3
4	Value41	Value43
5	Value51	Value53
6	Value61	Value63

	Variable 1
1	Value11
2	Value21
3	Value31
4	Value41
5	Value51
6	Value61

Figure 4.23 Inner Join Two Datasets with Some Common Variables

When merging DataFrames with some common variables, we may obtain two possible results: The output dataset contains either all available columns or only the common variables across all the DataFrames. In Figure 4.22, observation with row label 2 has only got values for `Variable 1` and `Variable 2`. As a result, the value for `Variable 3` of this row in the final DataFrame will be a missing value. This type of merging is called the *outer join*. But in Figure 4.23, the final DataFrame only consists of `Variable 1` since it is the only common variable in both DataFrames. This type of merging is called the *inner join*.

Similarly, we may also get multiple DataFrames with some common observations but totally different variables.

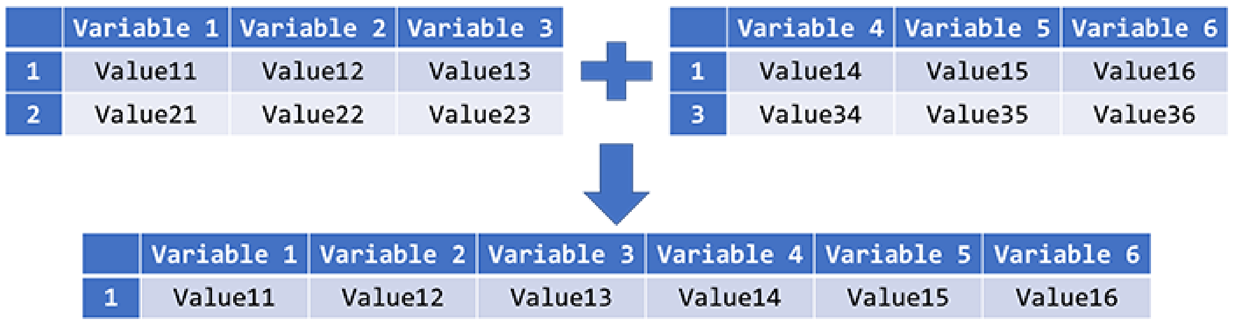


	Variable 1	Variable 2	Variable 3
1	Value11	Value12	Value13
2	Value21	Value22	Value23

	Variable 4	Variable 5	Variable 6
1	Value14	Value15	Value16
3	Value34	Value35	Value36

	Variable 1	Variable 2	Variable 3	Variable 4	Variable 5	Variable 6
1	Value11	Value12	Value13	Value14	Value15	Value16
2	Value21	Value22	Value23	NaN	NaN	NaN
3	NaN	NaN	NaN	Value34	Value35	Value36

Figure 4.24 Outer Join Two Datasets with Some Common Observations



	Variable 1	Variable 2	Variable 3
1	Value11	Value12	Value13
2	Value21	Value22	Value23

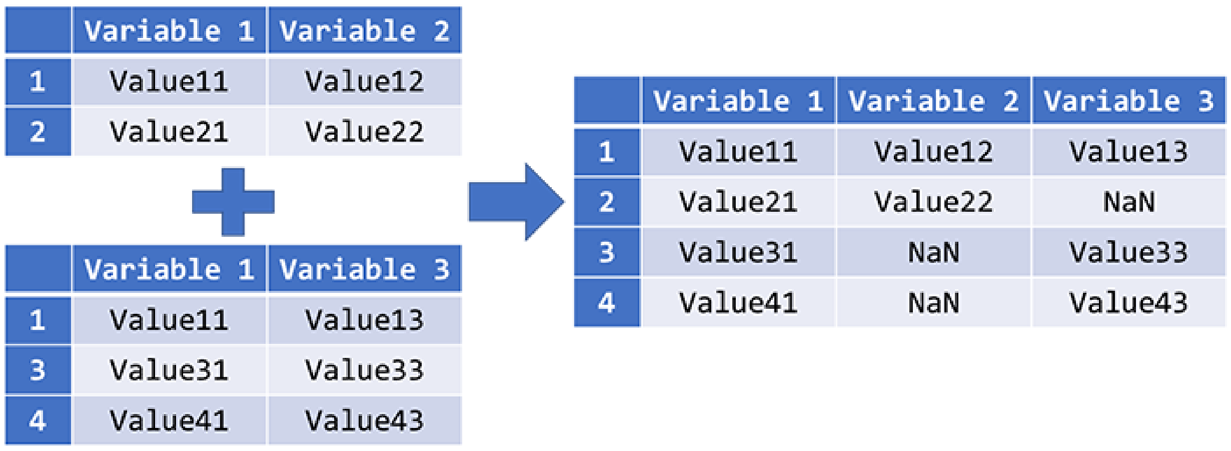
	Variable 4	Variable 5	Variable 6
1	Value14	Value15	Value16
3	Value34	Value35	Value36

	Variable 1	Variable 2	Variable 3	Variable 4	Variable 5	Variable 6
1	Value11	Value12	Value13	Value14	Value15	Value16

Figure 4.25 Inner Join Two Datasets with Some Common Observations

Same as in Figure 4.22 and Figure 4.23, we have two possible results here too: The output dataset contains either all available rows (outer join) or only the common rows across all the DataFrames (inner join). In Figure 4.24, observation with row label 2 has only got values for `Variable 1`, `Variable 2`, and `Variable 3`. As a result, the values for `Variable 4` to `Variable 6` of this row in the final DataFrame are entirely missing values. In Figure 4.25, the final DataFrame only consists of observation with `ID = 1` since it is the only common observation in both DataFrames.

In the last scenario, the multiple DataFrames to be merged have some common variables and observations. But there are also variables and observations that can only be found in either one of them. If we choose outer join to merge them, the result will be like the output dataset in Figure 4.26.



	Variable 1	Variable 2
1	Value11	Value12
2	Value21	Value22

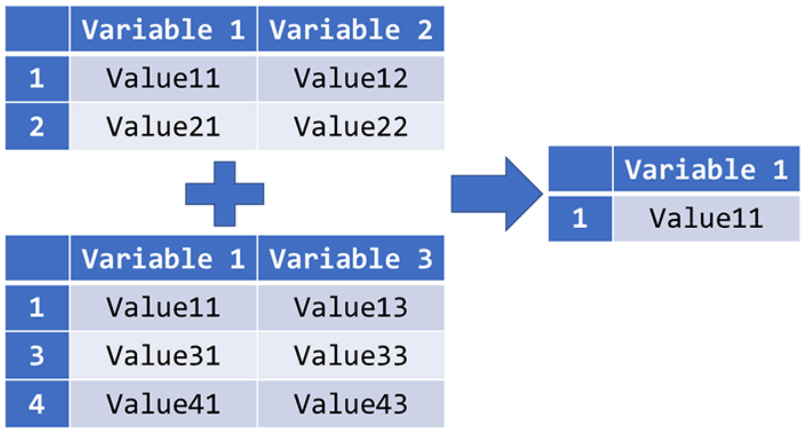
	Variable 1	Variable 3
1	Value11	Value13
3	Value31	Value33
4	Value41	Value43

	Variable 1	Variable 2	Variable 3
1	Value11	Value12	Value13
2	Value21	Value22	NaN
3	Value31	NaN	Value33
4	Value41	NaN	Value43

Figure 4.26 Outer Join Two Datasets with Different Shapes

The values of all available cells in either one of the original DataFrames will be taken over in the final DataFrame. Cells that were originally unavailable in both DataFrames such as `Value23` will become missing data.

On the other hand, if we choose inner join to merge them, the result will be like the output dataset in Figure 4.27.



	Variable 1	Variable 2
1	Value11	Value12
2	Value21	Value22

	Variable 1	Variable 3
1	Value11	Value13
3	Value31	Value33
4	Value41	Value43

	Variable 1
1	Value11

Figure 4.27 Inner Join Two Datasets with Different Shapes

Since `Value11` is the only common cell in both DataFrames, it will also be the only cell in the output DataFrame.

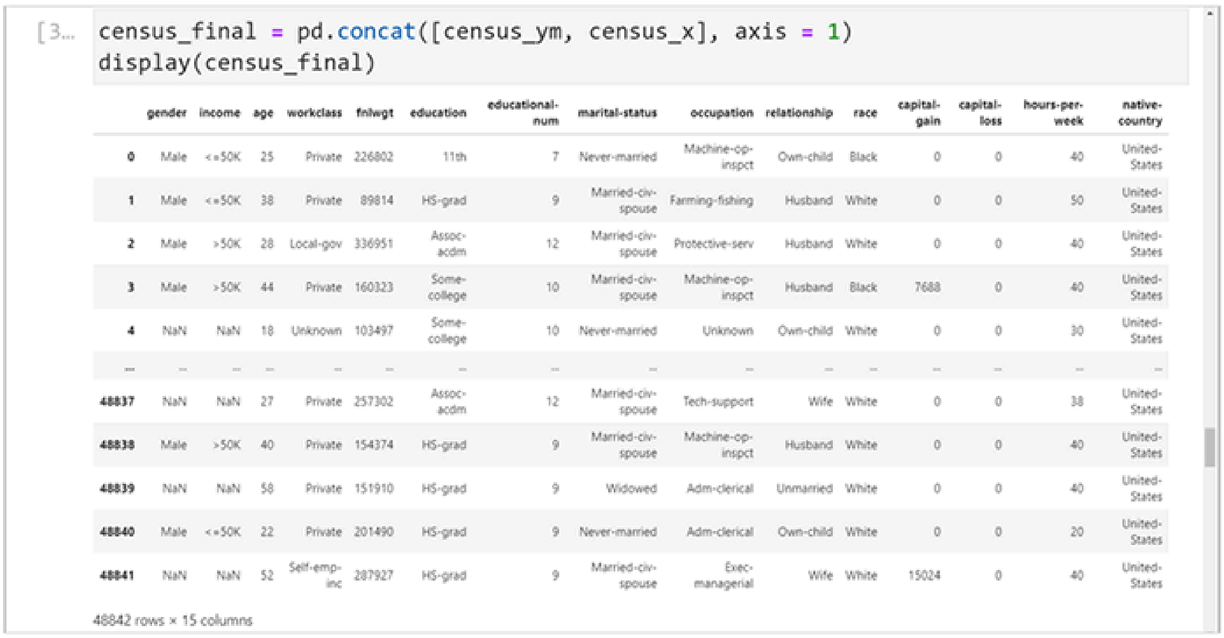
In Python, we can use the `concat()` function to merge multiple DataFrames in all the above-described scenarios. It is a rather complex method, and we will only list out the most commonly used parameters in our syntax introduction. For details, please refer to https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.concat.html.

```
finalDF_name = pd.concat(objs, axis, join)
```

The parameter `objs` is used for the specification of all the DataFrames to be concatenated. Be reminded that we need to put the names of the DataFrames in a list. The parameter `axis` is the direction along which the concatenation should take place. If `axis = 0`, the DataFrames will be concatenated below one another, and the concatenation will take place beside one another if `axis = 1`. The default value here is 0. With the join parameter, we can choose to carry out an outer join or inner join. The possible values here are `"outer"` and `"inner"`, written as string. If we omit this parameter, "outer" will be considered. The resulting DataFrame will be assigned to the object named `finalDF_name`.

Example (Cont'd): We have three sub-DataFrames of the US census study. The first one named `census_ym` contains all male observations. However, it has only two variables: `gender` and `income`. The second DataFrame named `census_yf` has the same variables, but it contains only female observations. The third DataFrame is called `census_x`. It contains all observations and all the variables of `census` except `income` and `gender`.

Suppose we concatenate `census_ym` and `census_x` first. The resulting DataFrame will contain all observations and all variables from the original census dataset.



```
[3... census_final = pd.concat([census_ym, census_x], axis = 1)
     display(census_final)
```

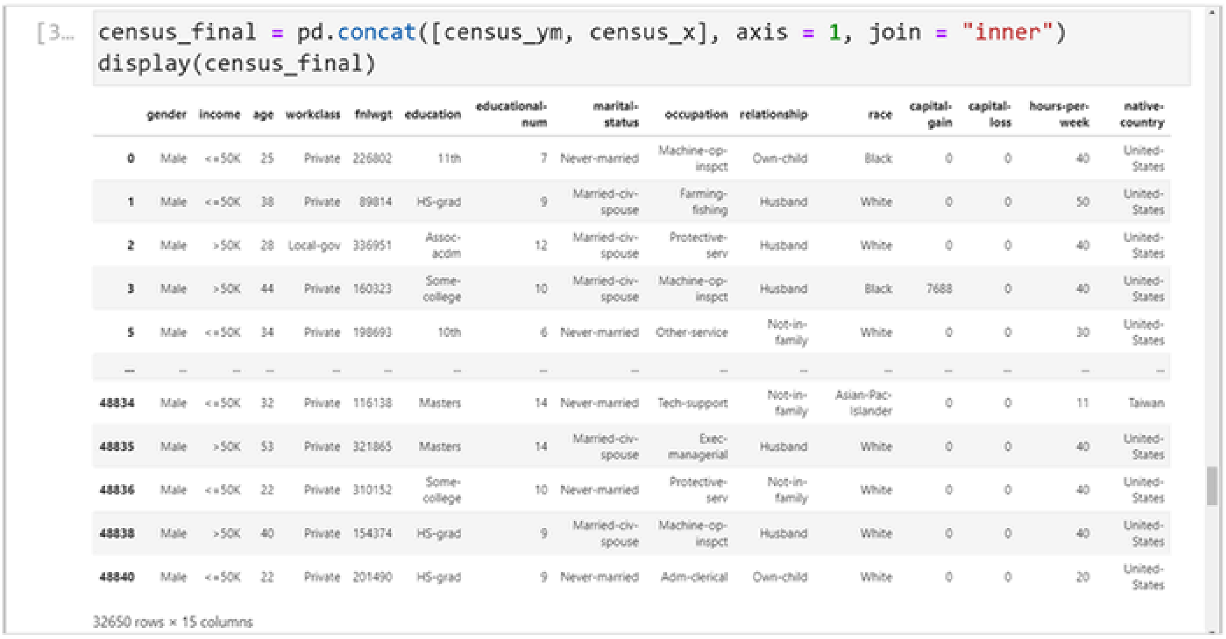
	gender	income	age	workclass	fnlwgt	education	educational-num	marital-status	occupation	relationship	race	capital-gain	capital-loss	hours-per-week	native-country
0	Male	<=50K	25	Private	226802	11th	7	Never-married	Machine-op-inspct	Own-child	Black	0	0	40	United-States
1	Male	<=50K	38	Private	89814	HS-grad	9	Married-civ-spouse	Farming-fishing	Husband	White	0	0	50	United-States
2	Male	>50K	28	Local-gov	336951	Assoc-acdm	12	Married-civ-spouse	Protective-serv	Husband	White	0	0	40	United-States
3	Male	>50K	44	Private	160323	Some-college	10	Married-civ-spouse	Machine-op-inspct	Husband	Black	7688	0	40	United-States
4	NaN	NaN	18	Unknown	103497	Some-college	10	Never-married	Unknown	Own-child	White	0	0	30	United-States
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
48837	NaN	NaN	27	Private	257302	Assoc-acdm	12	Married-civ-spouse	Tech-support	Wife	White	0	0	38	United-States
48838	Male	>50K	40	Private	154374	HS-grad	9	Married-civ-spouse	Machine-op-inspct	Husband	White	0	0	40	United-States
48839	NaN	NaN	58	Private	151910	HS-grad	9	Widowed	Adm-clerical	Unmarried	White	0	0	40	United-States
48840	Male	<=50K	22	Private	201490	HS-grad	9	Never-married	Adm-clerical	Own-child	White	0	0	20	United-States
48841	NaN	NaN	52	Self-emp-inc	287927	HS-grad	9	Married-civ-spouse	Exec-managerial	Wife	White	15024	0	40	United-States

48842 rows × 15 columns

Figure 4.28 Concatenating Two DataFrames with Different Shapes by Outer Join

Nevertheless, we can see in the fifth row, the value of the observation with row index 4 in the variable gender is `NaN`. We can conclude that this observation does not exist in `census_ym` since it is a female observation.

However, if we concatenate them with inner join, this observation will not exist in the final DataFrame `census_final`.



```
[3… census_final = pd.concat([census_ym, census_x], axis = 1, join = "inner")
    display(census_final)
```

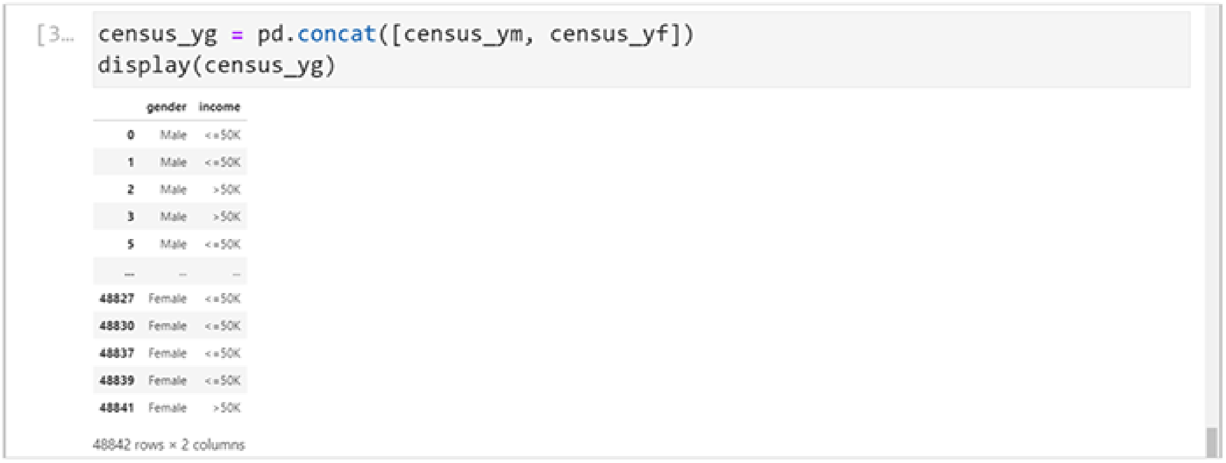
	gender	income	age	workclass	fnlwgt	education	educational-num	marital-status	occupation	relationship	race	capital-gain	capital-loss	hours-per-week	native-country
0	Male	<=50K	25	Private	226802	11th	7	Never-married	Machine-op-inspct	Own-child	Black	0	0	40	United-States
1	Male	<=50K	38	Private	89814	HS-grad	9	Married-civ-spouse	Farming-fishing	Husband	White	0	0	50	United-States
2	Male	>50K	28	Local-gov	336951	Assoc-acdm	12	Married-civ-spouse	Protective-serv	Husband	White	0	0	40	United-States
3	Male	>50K	44	Private	160323	Some-college	10	Married-civ-spouse	Machine-op-inspct	Husband	Black	7688	0	40	United-States
5	Male	<=50K	34	Private	198693	10th	6	Never-married	Other-service	Not-in-family	White	0	0	30	United-States
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
48834	Male	<=50K	32	Private	116138	Masters	14	Never-married	Tech-support	Not-in-family	Asian-Pac-Islander	0	0	11	Taiwan
48835	Male	>50K	53	Private	321865	Masters	14	Married-civ-spouse	Exec-managerial	Husband	White	0	0	40	United-States
48836	Male	<=50K	22	Private	310152	Some-college	10	Never-married	Protective-serv	Not-in-family	White	0	0	40	United-States
48838	Male	>50K	40	Private	154374	HS-grad	9	Married-civ-spouse	Machine-op-inspct	Husband	White	0	0	40	United-States
48840	Male	<=50K	22	Private	201490	HS-grad	9	Never-married	Adm-clerical	Own-child	White	0	0	20	United-States

32650 rows × 15 columns

Figure 4.29 Concatenating Two DataFrames with Different Shapes by Inner Join

Figure 4.29 shows that the observation with row index 4 is not included in the final DataFrame `census_final`. Furthermore, the number of rows here is 32,650, the number of rows in `census_ym`, instead of 48,842, the number of rows in `census_x`.

To reconstruct the original `census` DataFrame, we can first concatenate `census_ym` and `census_yf`.



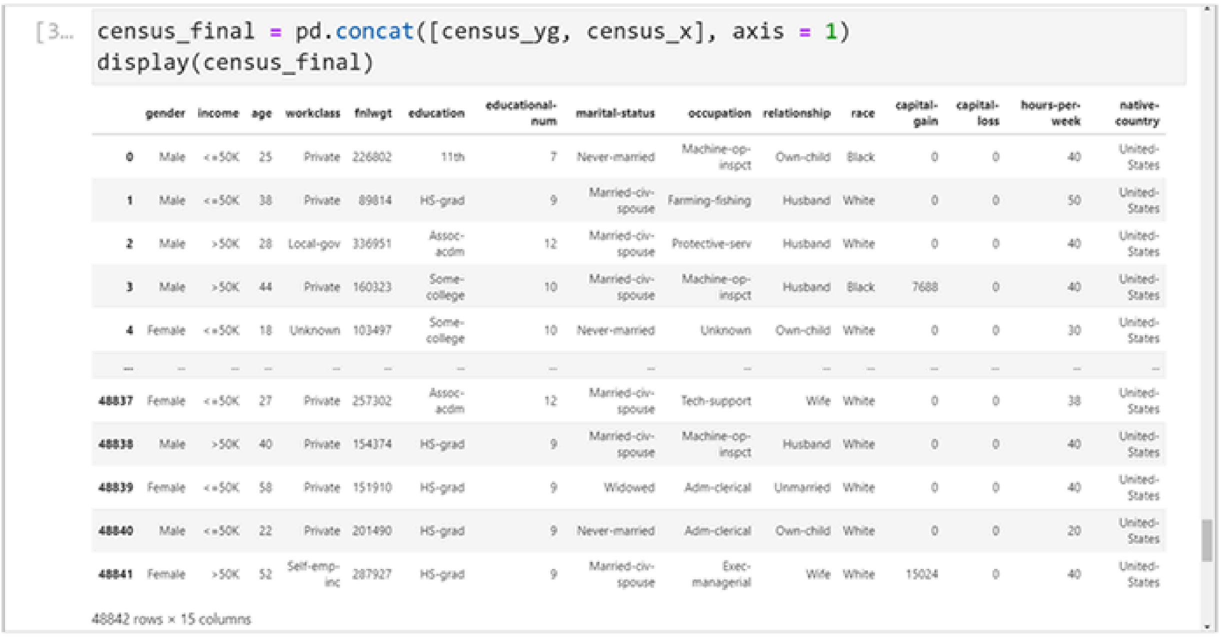
```
[3… census_yg = pd.concat([census_ym, census_yf])
    display(census_yg)
```

	gender	income
0	Male	<=50K
1	Male	<=50K
2	Male	>50K
3	Male	>50K
5	Male	<=50K
...	...	...
48827	Female	<=50K
48830	Female	<=50K
48837	Female	<=50K
48839	Female	<=50K
48841	Female	>50K

48842 rows × 2 columns

Figure 4.30 Concatenating Two DataFrames with Same Variables by Rows

We can see from the output that the resulting DataFrame `census_yg` has 48,842 rows, which corresponds to the number of rows in `census`. The DataFrame `census_yg` can now be merged with `census_x`.



```
[3...  census_final = pd.concat([census_yg, census_x], axis = 1)
       display(census_final)
```

	gender	income	age	workclass	fnlwgt	education	educational-num	marital-status	occupation	relationship	race	capital-gain	capital-loss	hours-per-week	native-country
0	Male	<=50K	25	Private	226802	11th	7	Never-married	Machine-op-inspct	Own-child	Black	0	0	40	United-States
1	Male	<=50K	38	Private	89814	HS-grad	9	Married-civ-spouse	Farming-fishing	Husband	White	0	0	50	United-States
2	Male	>50K	28	Local-gov	336951	Assoc-acdm	12	Married-civ-spouse	Protective-serv	Husband	White	0	0	40	United-States
3	Male	>50K	44	Private	160323	Some-college	10	Married-civ-spouse	Machine-op-inspct	Husband	Black	7688	0	40	United-States
4	Female	<=50K	18	Unknown	103497	Some-college	10	Never-married	Unknown	Own-child	White	0	0	30	United-States
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
48837	Female	<=50K	27	Private	257302	Assoc-acdm	12	Married-civ-spouse	Tech-support	Wife	White	0	0	38	United-States
48838	Male	>50K	40	Private	154374	HS-grad	9	Married-civ-spouse	Machine-op-inspct	Husband	White	0	0	40	United-States
48839	Female	<=50K	58	Private	151910	HS-grad	9	Widowed	Adm-clerical	Unmarried	White	0	0	40	United-States
48840	Male	<=50K	22	Private	201490	HS-grad	9	Never-married	Adm-clerical	Own-child	White	0	0	20	United-States
48841	Female	>50K	52	Self-emp-inc	287927	HS-grad	9	Married-civ-spouse	Exec-managerial	Wife	White	15024	0	40	United-States

```
48842 rows × 15 columns
```

Figure 4.31 Concatenating Two DataFrames with Same Observations by Columns

As we can see, the rows in the final DataFrame `census_final` are sorted by the row indices automatically.

Read

Refer to the three links below for more details and examples on merging DataFrames using the pandas package:

https://pandas.pydata.org/pandas-docs/stable/user_guide/merging.html

Refer to the three links below for more details and examples on concatenating DataFrames using the `concat()` function of the pandas package:

https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.concat.html

Chapter 4: Missing Data and Outliers

Lesson Recording

Missing Data and Outliers in pandas DataFrames

In empirical studies, it often occurs that an observed value of a variable is missing. There are many reasons for missing data: defective measurement tools, withdrawal from the study, refusal of responses to sensitive questions, etc. In Python, there is the `NoneType` to indicate missing data. Different packages have different ways to display a missing value. For instance, pandas uses a special floating-point value for missing values, and NumPy uses `NaN` which stands for "Not a Number".

Missing data are not desirable for data analytics since they cannot be included in constructing models, forecasting, etc. Statistical estimation of parameters can be biased. In pandas, when we use statistical functions on DataFrames, missing values are typically ignored by these functions. As a result, the execution of the code will not be interrupted, but the computation of these functions could be due to unequal underlying sample sizes for each variable.

4.1 Identifying Missing Values

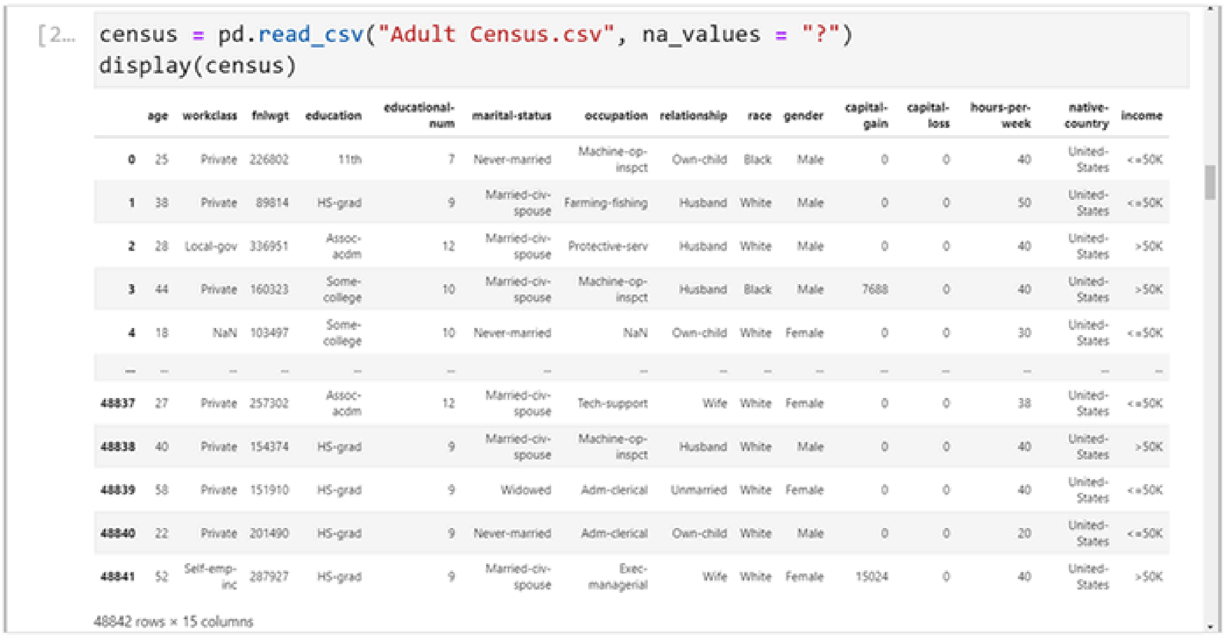
Most of the time, we have to work with datasets provided from external sources, and missing values can be referred very differently. The reasons of such discrepancies could be typing errors, or the varying habit of the data collectors when entering missing values, or the limitation of the software used for data entry, etc. In pandas, readers such as the `read_csv()` function provide two parameters, `na_filter` and `na_values`, to convert certain strings to missing values directly while the data are being converted to pandas DataFrame.

```
DataFrame_name = pd.read_csv("csv_file_name.csv", na_values
 =
                              "na_string", na_filer = True/
False)
```

The default value of the `na_filter` parameter is `True`. In this case, pandas will convert all white spaces `""` to `NaN`. However, there could be situations where white space is an actual value of interest and not a missing value. The filter should then be turned off and the value would be `False`.

With the parameter `na_values`, we can declare certain strings from our DataFrame to be recognised as missing values. By default, strings like `""`, `"#N/A"`, `"#N/A N/A"`, `"#NA"`, `"-1.#IND"`, `"-1.#QNAN"`, `"-NaN"`, `"-nan"`, `"1.#IND"`, `"1.#QNAN"`, `"N/A"`, `"NA"`, `"NULL"`, `"NaN"`, `"n/a"`, `"nan"`, `"null"` are treated as missing values and do not need to be specified explicitly with this parameter.

Example (Cont'd): From Figure 4.1 and Figure 4.2, we can recognise that question marks are used to indicate missing values in the US Adult Census dataset. Suppose we would like to declare every cell that contains a question mark solely as a missing value.



```
[2...  census = pd.read_csv("Adult Census.csv", na_values = "?")
       display(census)
```

	age	workclass	fnlwgt	education	educational-num	marital-status	occupation	relationship	race	gender	capital-gain	capital-loss	hours-per-week	native-country	income
0	25	Private	226802	11th	7	Never-married	Machine-op-inspct	Own-child	Black	Male	0	0	40	United-States	<=50K
1	38	Private	89814	HS-grad	9	Married-civ-spouse	Farming-fishing	Husband	White	Male	0	0	50	United-States	<=50K
2	28	Local-gov	336951	Assoc-acdm	12	Married-civ-spouse	Protective-serv	Husband	White	Male	0	0	40	United-States	>50K
3	44	Private	160323	Some-college	10	Married-civ-spouse	Machine-op-inspct	Husband	Black	Male	7688	0	40	United-States	>50K
4	18	NaN	103497	Some-college	10	Never-married	NaN	Own-child	White	Female	0	0	30	United-States	<=50K
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
48837	27	Private	257302	Assoc-acdm	12	Married-civ-spouse	Tech-support	Wife	White	Female	0	0	38	United-States	<=50K
48838	40	Private	154374	HS-grad	9	Married-civ-spouse	Machine-op-inspct	Husband	White	Male	0	0	40	United-States	>50K
48839	58	Private	151910	HS-grad	9	Widowed	Adm-clerical	Unmarried	White	Female	0	0	40	United-States	<=50K
48840	22	Private	201490	HS-grad	9	Never-married	Adm-clerical	Own-child	White	Male	0	0	20	United-States	<=50K
48841	52	Self-emp-inc	287927	HS-grad	9	Married-civ-spouse	Exec-managerial	Wife	White	Female	15024	0	40	United-States	>50K

48842 rows × 15 columns

Figure 4.32 Declaring Specific Strings as Missing Values While Importing Data

In the `read_csv()` function, we specify a single question mark as string that should be identified as a missing value in the `census` DataFrame. In Figure 4.32, we can see that the values for `workclass` and `occupation` in the fifth row are now `NaN`, whereas in Figure 4.1, they were simply `"?"`.

Read

Refer to the link below for more details and examples on the parameters associated to missing values in the `read_csv()` function of the pandas package:

https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.read_csv.html

4.2 Locating Missing Values

In Chapter 4.1, we learned that the parameters in the `read_csv()` function can instruct Python to indicate missing values clearly with `NaN` in the DataFrame. Though they will become uniquely identifiable, it is neither easy to locate their positions, nor to detect their existence, if the dataset contains a large number of rows and columns. One way to find out their existence and positions is to count the `NaN`s in each row and each column.

If the number of `NaN`s in a column is larger than zero, we have then identified the variables in which missing values exist and including these variables may create biasedness in our analytics tasks. And if the number of `NaN`s in a variable is large, we can also conclude that the variable may not contain sufficient data for reliable analyses. Equivalently, we can apply the same approach to rows. If the number of `NaN`s in a row is large, we know that missing values do not only exist for this observation, it may also not be carrying much information for our analyses.

```
DataFrame_name.isnull().sum(axis = 0)
DataFrame_name.isnull().sum(axis = 1)
```

The above syntax is in fact a Boolean masking. It contains two methods of the pandas package. The `.isnull()` method instructs Python to check every cell of the DataFrame and then return `True` if it is an `NaN`. Subsequently, Python should return the sum of each row or each column of the Boolean mask. If the parameter axis is set to 0, the values in a column will be added up together. And if `axis = 1`, we will obtain the sum of the row instead. The default axis here is 0. Since `True` is usually represented by 1 and `False` by 0 when converting a Boolean variable to a numeric value, the sum of a row or a column with only Boolean values will therefore be the same as counting the occurrence of `True` in it.

If our intension is just to check the existence of missing values, we can use the `.any()` method instead. The `.any()` method will return `True` if at least one of the elements in the array returned by the `.isnull()` method is `True`.

```
DataFrame_name.isnull().any(axis = 0)
DataFrame_name.isnull().any(axis = 1)
```

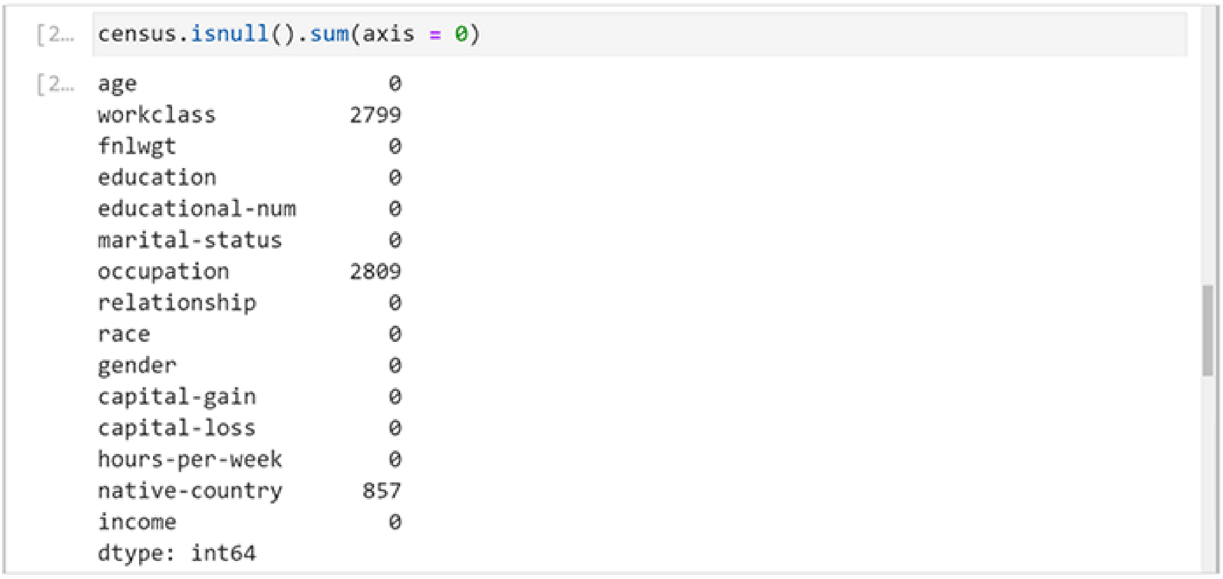
We can retrieve the indices of the rows or columns with missing data by applying the `.index` method on the resulting object from the syntax above.

```
object_name = DataFrame_name.isnull().any()
object_name[object_name == True].index
```

Counting the `NaN`s in columns has actually a different meaning than counting them in rows. When we count the number of `NaN`s in columns, we are checking on the existence of missing values in each variable. If they exist, we may need different approaches to adjust the data for different types of variable. For instance, if they exist in a numeric variable, we can replace the missing values by zero or by the mean of the variables. And if a text variable contains missing data, we may add a response category such as “no reply” to it. We can also choose to neglect them if the variable is irrelevant for our analyses of the data.

By counting the `NaN`s in rows, however, we intend to identify those observations with missing values in at least one of the variables. Depending on the analyses and the importance of the observation, we can choose to delete the observation or to apply the appropriate data adjustments to the affected columns.

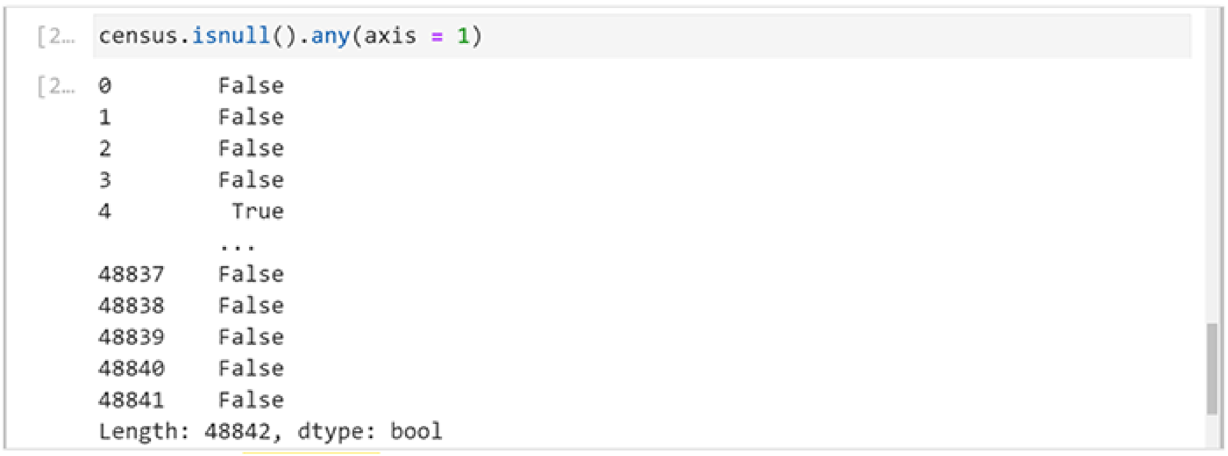
Example (Cont'd): From Figure 4.32, we can identify two missing values in `workclass` and `occupation` for the fifth observation. Now we would like to find out whether there are more missing values in these two and other variables.



```
[2… census.isnull().sum(axis = 0)

[2… age                  0
    workclass         2799
    fnlwgt               0
    education            0
    educational-num      0
    marital-status       0
    occupation        2809
    relationship         0
    race                 0
    gender               0
    capital-gain         0
    capital-loss         0
    hours-per-week       0
    native-country     857
    income               0
    dtype: int64
```

Figure 4.33 Counting the Number of Missing Values in Each Variable

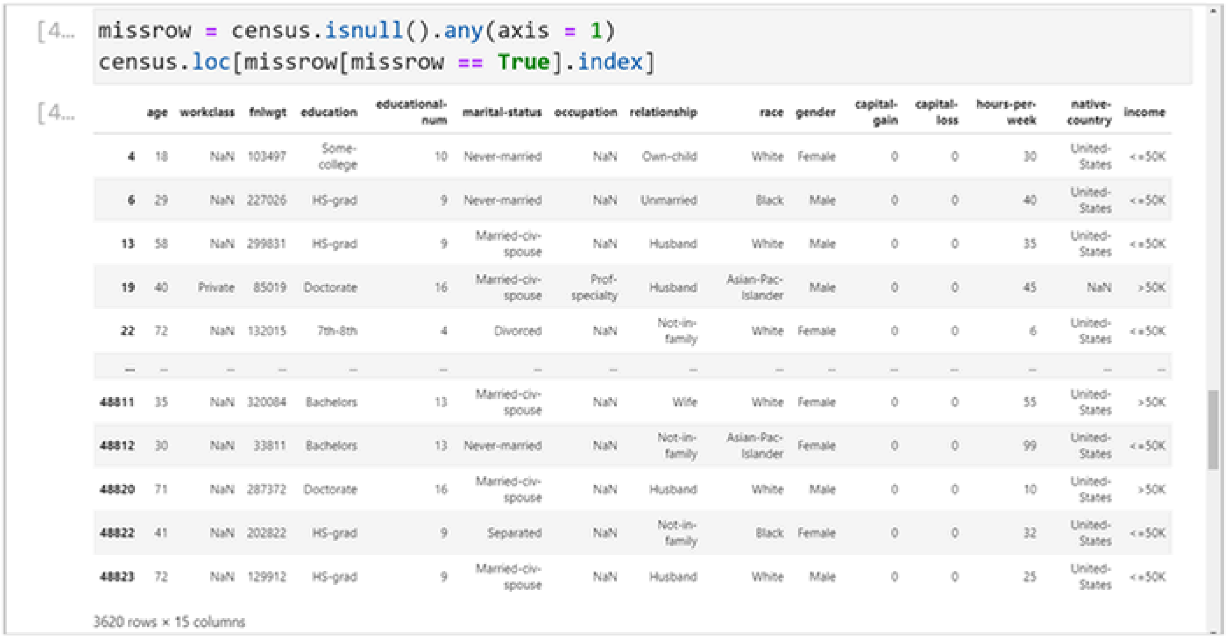
The output shows that `workclass`, `occupation`, and `native-country` are the three variables with missing data. Their proportions of missing data are 2,799 (5.7%), 2,809 (5.7%) and 857 (1.8%) out of 48,842 observations, respectively. We need to further study these observations to conclude on the adjustment we shall apply on the missing values. For this purpose, we shall find out all the observations, or their row indices, with at least one missing value.



```
[2… census.isnull().any(axis = 1)

[2… 0        False
    1        False
    2        False
    3        False
    4         True
             ...
    48837    False
    48838    False
    48839    False
    48840    False
    48841    False
    Length: 48842, dtype: bool
```

Figure 4.34 Identifying Observations with Missing Values

The output generated in Figure 4.34 is rather not satisfactory since we only see the returned Boolean value from the chained methods `.isnull().any(axis = 1)`. Furthermore, it is only useful to us if the rows with missing values are selected from the DataFrame. As a result, we need to filter `census` with the results above.



```
[4… missrow = census.isnull().any(axis = 1)
    census.loc[missrow[missrow == True].index]
```

	age	workclass	fnlwgt	education	educational-num	marital-status	occupation	relationship	race	gender	capital-gain	capital-loss	hours-per-week	native-country	income
4	18	NaN	103497	Some-college	10	Never-married	NaN	Own-child	White	Female	0	0	30	United-States	<=50K
6	29	NaN	227026	HS-grad	9	Never-married	NaN	Unmarried	Black	Male	0	0	40	United-States	<=50K
13	58	NaN	299831	HS-grad	9	Married-civ-spouse	NaN	Husband	White	Male	0	0	35	United-States	<=50K
19	40	Private	85019	Doctorate	16	Married-civ-spouse	Prof-specialty	Husband	Asian-Pac-Islander	Male	0	0	45	NaN	>50K
22	72	NaN	132015	7th-8th	4	Divorced	NaN	Not-in-family	White	Female	0	0	6	United-States	<=50K
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
48811	35	NaN	320084	Bachelors	13	Married-civ-spouse	NaN	Wife	White	Female	0	0	55	United-States	>50K
48812	30	NaN	33811	Bachelors	13	Never-married	NaN	Not-in-family	Asian-Pac-Islander	Female	0	0	99	United-States	<=50K
48820	71	NaN	287372	Doctorate	16	Married-civ-spouse	NaN	Husband	White	Male	0	0	10	United-States	>50K
48822	41	NaN	202822	HS-grad	9	Separated	NaN	Not-in-family	Black	Female	0	0	32	United-States	<=50K
48823	72	NaN	129912	HS-grad	9	Married-civ-spouse	NaN	Husband	White	Male	0	0	25	United-States	<=50K

3620 rows × 15 columns

Figure 4.35 Selecting Observations with Missing Values from a DataFrame

First, we save the output generated in Figure 4.34 as an object named `missrow`. After that, we use a Boolean mask to select only those "`True`" observations from `missrow`, and the `.index` method will then return the corresponding row indices to us for selection. Eventually, we apply these row indices to subset `census`. If we wish to work on this subset of the DataFrame further, we can also assign it to an object in the second line of our code.

Taken from Figure 4.35, there are a total of 3620 rows with at least one missing value in `workclass`, `occupation`, and `native-country`. We can now study the data of these observations and decide on the adjustment measures subsequently.

Read

Refer to the link below for more details and examples on the `.isnull()` method of the pandas package:

https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.isnull.html

Refer to the link below for more details and examples on the `.sum()` method of the pandas package:

https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.sum.html

Refer to the link below for more details and examples on the `.any()` method of the pandas package:

https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.any.html

Refer to the link below for the `index()` method of the pandas package:

https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.index.html

4.3 Replacing Missing Values

After checking the existence of missing values in a DataFrame and locating them, we should decide on how to deal with them. Usually, we can consider deleting the entire observations, replacing them by other values, or simply ignoring them.

To delete an entire row with missing values from the DataFrame, we have two options: the `.drop()` and `.dropna()` methods.

```
DataFrame_name.drop(axis = 0, index = [index1, index2, …])
```

With the `.drop()` method, we can delete an entire row or column by specifying the corresponding indices resulting from the localisation methods introduced in Chapter 4.2. The parameter `axis` indicates whether rows (0) or columns (1) should be dropped.

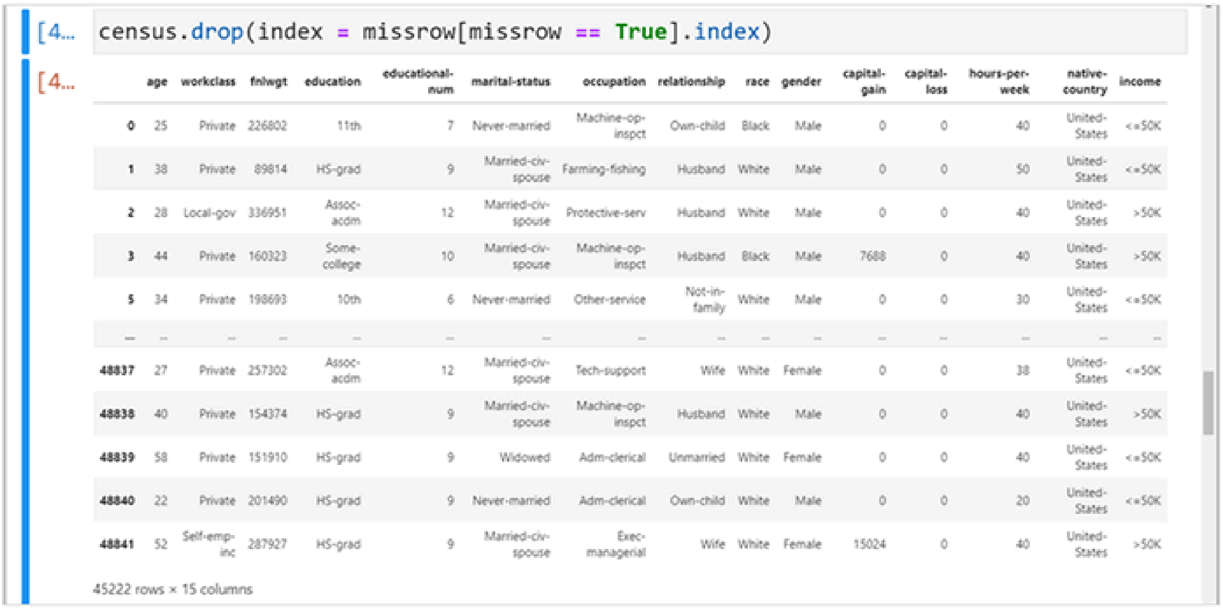
The `.dropna()` method combines the localisation and removal of rows or columns with missing data in a single function. Its usage is rather convenient since we can omit using the `.isnull().any()` and `.index()` methods before dropping the corresponding observations or variables.

```
DataFrame_name.dropna(axis = 0, how = "any"/"all")
```

The `axis` parameter in pandas functions or methods should not be a stranger to us anymore. With the `how` parameter, however, we can instruct Python to drop an observation with only missing values in all variables (`all`), or to drop an observation with at least one missing value in any variable (`any`).

The drawback of the `.dropna()` method is the equal treatment for all missing values throughout the entire dataset. As mentioned in the previous chapters, we have multiple ways to adjust missing data for different types of variable. And depending on the observed values of other variables, we may also want to keep some of the rows with missing data while deleting others.

Example (Cont'd): Now we would like to remove all the rows with missing values using the `.drop()` method.



```
[4… census.drop(index = missrow[missrow == True].index)
[4…
```

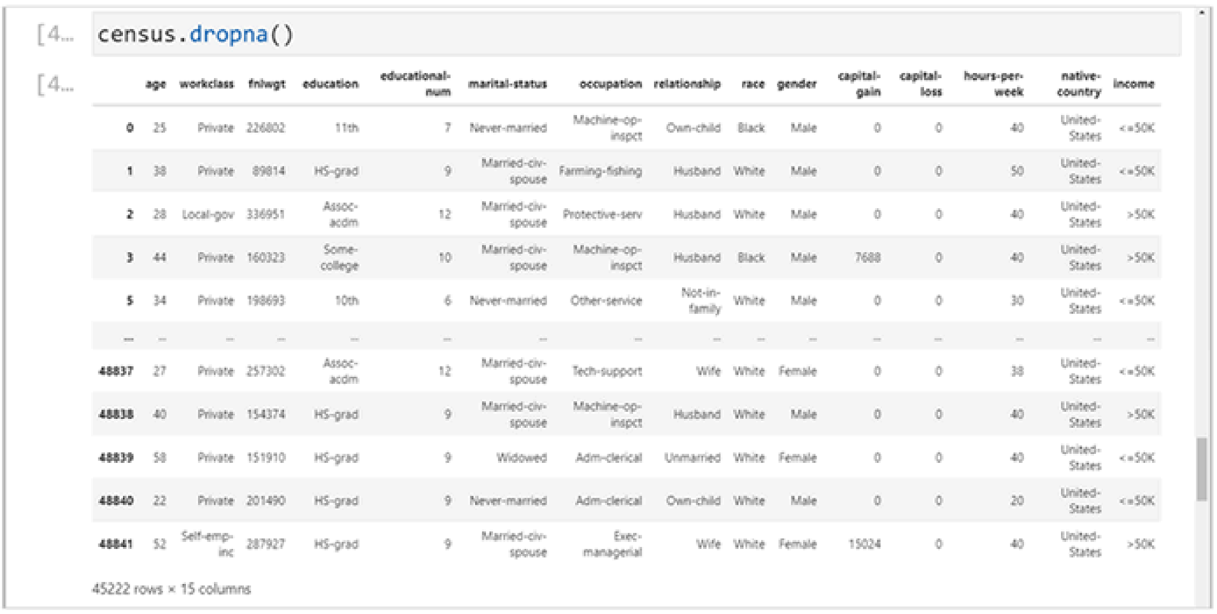
	age	workclass	fnlwgt	education	educational-num	marital-status	occupation	relationship	race	gender	capital-gain	capital-loss	hours-per-week	native-country	income
0	25	Private	226802	11th	7	Never-married	Machine-op-inspct	Own-child	Black	Male	0	0	40	United-States	<=50K
1	38	Private	89814	HS-grad	9	Married-civ-spouse	Farming-fishing	Husband	White	Male	0	0	50	United-States	<=50K
2	28	Local-gov	336951	Assoc-acdm	12	Married-civ-spouse	Protective-serv	Husband	White	Male	0	0	40	United-States	>50K
3	44	Private	160323	Some-college	10	Married-civ-spouse	Machine-op-inspct	Husband	Black	Male	7688	0	40	United-States	>50K
5	34	Private	198693	10th	6	Never-married	Other-service	Not-in-family	White	Male	0	0	30	United-States	<=50K
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
48837	27	Private	257302	Assoc-acdm	12	Married-civ-spouse	Tech-support	Wife	White	Female	0	0	38	United-States	<=50K
48838	40	Private	154374	HS-grad	9	Married-civ-spouse	Machine-op-inspct	Husband	White	Male	0	0	40	United-States	>50K
48839	58	Private	151910	HS-grad	9	Widowed	Adm-clerical	Unmarried	White	Female	0	0	40	United-States	<=50K
48840	22	Private	201490	HS-grad	9	Never-married	Adm-clerical	Own-child	White	Male	0	0	20	United-States	<=50K
48841	52	Self-emp-inc	287927	HS-grad	9	Married-civ-spouse	Exec-managerial	Wife	White	Female	15024	0	40	United-States	>50K

45222 rows × 15 columns

Figure 4.36 Dropping Observations with Missing Values Using `.drop()` Method

The `missrow` object is still the same one created in Figure 4.35, and the indices assigned to the `index` parameter in the `.drop()` method are determined by the same syntax used in the index operator for `missrow` in the same figure.

We can obtain the same result using the `.dropna()` function.



```
[4... census.dropna()
```

	age	workclass	fnlwgt	education	educational-num	marital-status	occupation	relationship	race	gender	capital-gain	capital-loss	hours-per-week	native-country	income
0	25	Private	226802	11th	7	Never-married	Machine-op-inspct	Own-child	Black	Male	0	0	40	United-States	<=50K
1	38	Private	89814	HS-grad	9	Married-civ-spouse	Farming-fishing	Husband	White	Male	0	0	50	United-States	<=50K
2	28	Local-gov	336951	Assoc-acdm	12	Married-civ-spouse	Protective-serv	Husband	White	Male	0	0	40	United-States	>50K
3	44	Private	160323	Some-college	10	Married-civ-spouse	Machine-op-inspct	Husband	Black	Male	7688	0	40	United-States	>50K
5	34	Private	198693	10th	6	Never-married	Other-service	Not-in-family	White	Male	0	0	30	United-States	<=50K
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
48837	27	Private	257302	Assoc-acdm	12	Married-civ-spouse	Tech-support	Wife	White	Female	0	0	38	United-States	<=50K
48838	40	Private	154374	HS-grad	9	Married-civ-spouse	Machine-op-inspct	Husband	White	Male	0	0	40	United-States	>50K
48839	58	Private	151910	HS-grad	9	Widowed	Adm-clerical	Unmarried	White	Female	0	0	40	United-States	<=50K
48840	22	Private	201490	HS-grad	9	Never-married	Adm-clerical	Own-child	White	Male	0	0	20	United-States	<=50K
48841	52	Self-emp-inc	287927	HS-grad	9	Married-civ-spouse	Exec-managerial	Wife	White	Female	15024	0	40	United-States	>50K

45222 rows × 15 columns

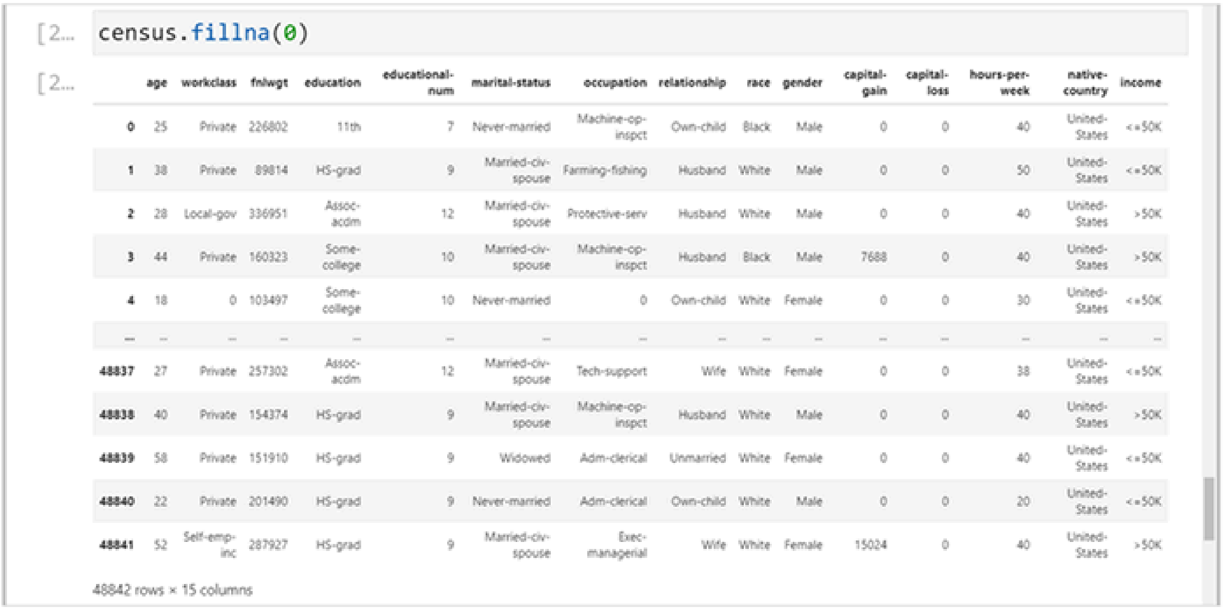
Figure 4.37 Dropping Observations with Missing Values Using `.dropna()` Method

Another possibility in dealing with missing values is to replace them by a pre-defined value. The most common values used for such purpose are 0 or the variable mean. Some literatures also suggest more sophisticated approaches such as interpolation, extrapolation, or estimation. In Python, the pandas package facilitates replacement of missing values by the `.fillna()` method.

```
DataFrame_name.fillna(value = repl_value)
DataFrame_name["column_label"].fillna(value = repl_value)
```

Basically, if we apply the `.fillna()` method on the entire DataFrame, it will replace all missing values that Python could find with the value specified in the parameter. But if we specify a column in the DataFrame and attach the `.fillna()` method to it, only the missing values found in the corresponding variable will be replaced. By doing this, we can treat missing data in various variable types differently.

Example (Cont'd): Suppose we decide to simply replace all missing values in the DataFrame `census` by 0.



```
[2... census.fillna(0)
```

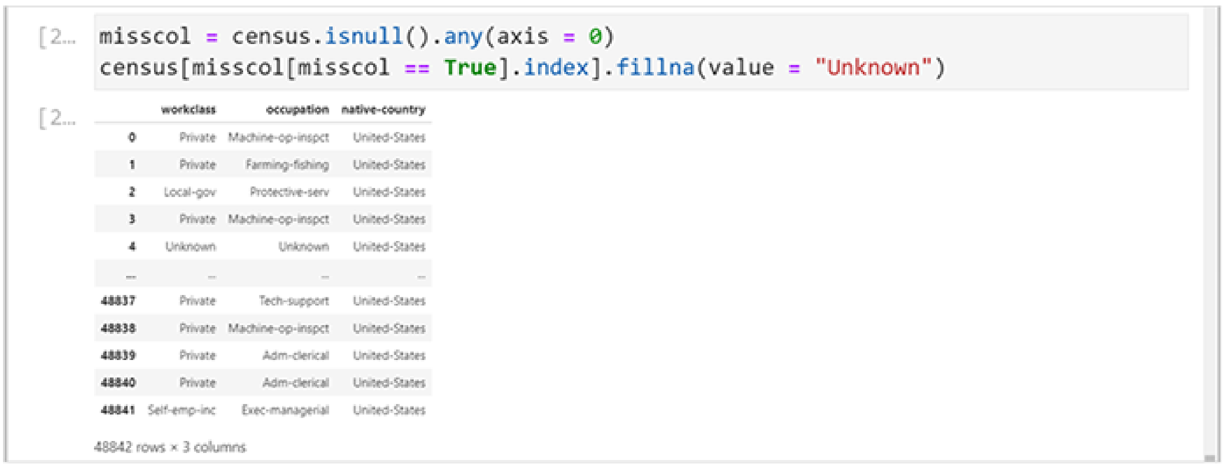
[2...

	age	workclass	fnlwgt	education	educational-num	marital-status	occupation	relationship	race	gender	capital-gain	capital-loss	hours-per-week	native-country	income
0	25	Private	226802	11th	7	Never-married	Machine-op-inspct	Own-child	Black	Male	0	0	40	United-States	<=50K
1	38	Private	89814	HS-grad	9	Married-civ-spouse	Farming-fishing	Husband	White	Male	0	0	50	United-States	<=50K
2	28	Local-gov	336951	Assoc-acdm	12	Married-civ-spouse	Protective-serv	Husband	White	Male	0	0	40	United-States	>50K
3	44	Private	160323	Some-college	10	Married-civ-spouse	Machine-op-inspct	Husband	Black	Male	7688	0	40	United-States	>50K
4	18	0	103497	Some-college	10	Never-married	0	Own-child	White	Female	0	0	30	United-States	<=50K
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
48837	27	Private	257302	Assoc-acdm	12	Married-civ-spouse	Tech-support	Wife	White	Female	0	0	38	United-States	<=50K
48838	40	Private	154374	HS-grad	9	Married-civ-spouse	Machine-op-inspct	Husband	White	Male	0	0	40	United-States	>50K
48839	58	Private	151910	HS-grad	9	Widowed	Adm-clerical	Unmarried	White	Female	0	0	40	United-States	<=50K
48840	22	Private	201490	HS-grad	9	Never-married	Adm-clerical	Own-child	White	Male	0	0	20	United-States	<=50K
48841	52	Self-emp-inc	287927	HS-grad	9	Married-civ-spouse	Exec-managerial	Wife	White	Female	15024	0	40	United-States	>50K

48842 rows × 15 columns

Figure 4.38 Replacing Missing Values by 0 in the Entire DataFrame

The output shows that all the missing data (`NaN`) in row 5 are now replaced by 0. Nevertheless, it looks rather odd to have a value 0 in the variables `workclass` and `occupation`. As a result, instead of replacing them by 0, we would rather replacing them with the string "`Unknown`".



```
[2… misscol = census.isnull().any(axis = 0)
    census[misscol[misscol == True].index].fillna(value = "Unknown")
```

```
[2…        workclass        occupation  native-country
        0    Private  Machine-op-inspct  United-States
        1    Private    Farming-fishing  United-States
        2  Local-gov    Protective-serv  United-States
        3    Private  Machine-op-inspct  United-States
        4    Unknown            Unknown  United-States
      ...        ...                ...            ...
    48837    Private       Tech-support  United-States
    48838    Private  Machine-op-inspct  United-States
    48839    Private       Adm-clerical  United-States
    48840    Private       Adm-clerical  United-States
    48841  Self-emp-inc  Exec-managerial  United-States

    48842 rows × 3 columns
```

Figure 4.39 Replacing Missing Values by "Unknown" in Specific Columns

In the first line, we apply the same chained methods `.isnull().any(axis = 0)` on the DataFrame `census` to detect the existence of missing values in each column. In the second line, we select only those columns where the returned values from the first line are `True` using the `.index` method. We then apply the `.fillna()` method to replace the missing values by "`Unknown`", which is the replacement string assigned to the parameter `value`.

Read

Refer to the link below for more details and examples on the `.drop()` method of the pandas package:

https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.drop.html

Refer to the link below for more details and examples on the `.dropna()` method of the pandas package:

https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.dropna.html

Refer to the link below for more details and examples on the `.fillna()` method of the pandas package:

https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.fillna.html

4.4 Detecting and Removing Outliers

Beside missing data, outliers are data that may cause biasedness in the estimation of statistical parameters and hence the goodness of fit of the models. Since biased estimates are undesirable, it is important to identify them and undertake appropriate adjustments before conducting any analysis.

Basically, we can use statistics such as the interquartile range (IQR) to detect the existence of outliers in a variable. Furthermore, visualisation like boxplots or histogram can also be useful to examine the distribution of the variables.

In Chapter 3.2 of Study Unit 3, we have learned how to use the sub-package matplotlib.pyplot to draw histogram in Python. The `boxplot()` function from the same sub-package facilitates the creation of boxplots for outlier detection.

To compute the interquartile range, we can use the `.quantile()` method to determine the first and third quartiles of the variable.

```
DataFrame_name["column_label"].quantile(q = quantile)
```

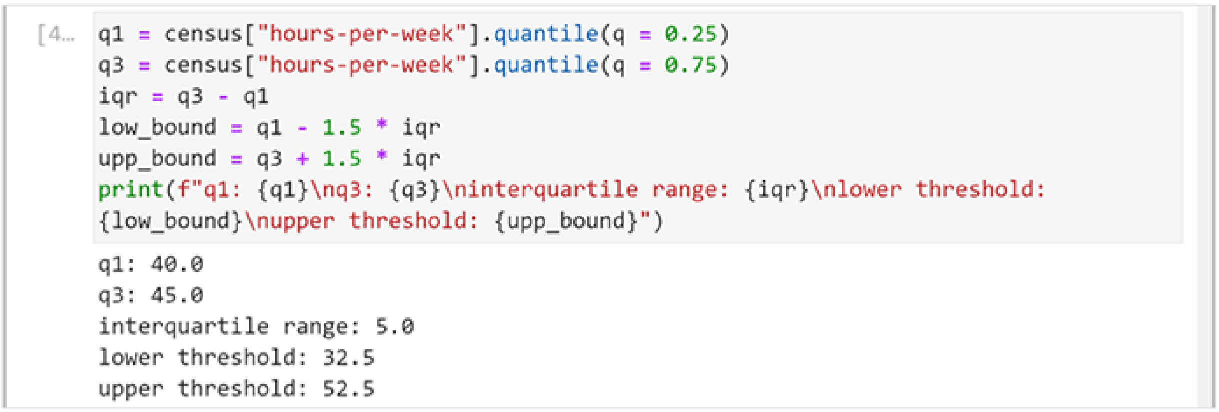
With the parameter `q`, which is a value between 0 and 1, we can define the quantile of the distribution that the `.quantile()` method should return to us. Once the 0.25 and 0.75 quantiles of the target variable is obtained, the interquartile range `iqr` can be computed by `iqr = q3 - q1`. An observation `y` is considered as outlier if `y < q1 - 1.5 * iqr` or `y > q3 + 1.5 * iqr`.

The usual practice in dealing with outliers is to remove them from the dataset. In Python, it suffices to keep observations that do not contain outliers in the target variable. The syntax below generates a subset of rows that do not fulfil the above outlier condition.

```
DF[~((DF["Col"] < q1 – 1.5 * iqr) | (DF["Col"] > q3 + 1.5 *
 iqr))]
```

Note that `DF` represents the `DataFrame_Name` and `Col` is the `column_label`. The condition left from the bitwise or operator "`|`" selects all observations with values in "`Col`" smaller than `q1 – 1.5 * iqr` whereas the condition right from it selects those observations larger than `q3 + 1.5 * iqr`. Nevertheless, this would be the combined condition to select all the outliers. To invert the selection, we need to put the bitwise not operator "~" before the entire condition, which must then be put in a pair of parentheses.

Example (Cont'd): In the following, we will use the aforementioned interquartile range rule to detect outliers in the variable `hours-per-week`, i.e. observations with extraordinary high or low number of working hours.

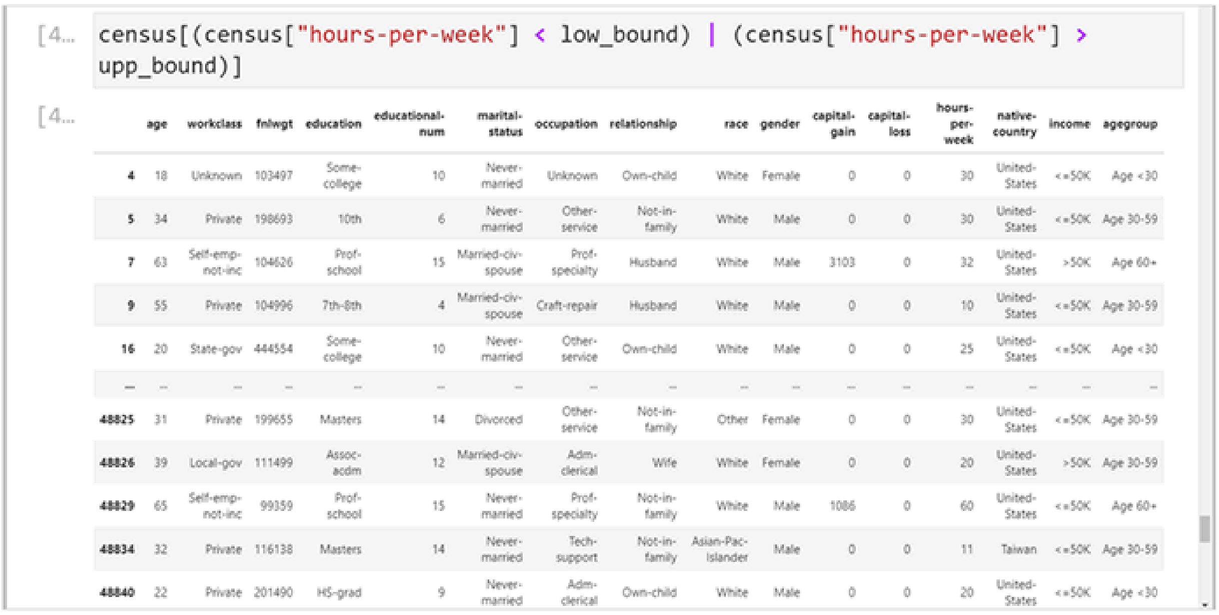


```
[4…  q1 = census["hours-per-week"].quantile(q = 0.25)
     q3 = census["hours-per-week"].quantile(q = 0.75)
     iqr = q3 - q1
     low_bound = q1 - 1.5 * iqr
     upp_bound = q3 + 1.5 * iqr
     print(f"q1: {q1}\nq3: {q3}\ninterquartile range: {iqr}\nlower threshold:
     {low_bound}\nupper threshold: {upp_bound}")

     q1: 40.0
     q3: 45.0
     interquartile range: 5.0
     lower threshold: 32.5
     upper threshold: 52.5
```

Figure 4.40 Computing Criteria for Outlier Detection in a Numeric Variable

Based on the results in Figure 4.40, half of the sample works between 40 and 45 hours weekly on average. The corresponding upper and lower thresholds to differentiate outliers from "normal" data are 32.5 and 52.5, respectively.

In the next step, we can select those outlier observations for checking before dropping them from the DataFrame eventually.



```
[4… census[(census["hours-per-week"] < low_bound) | (census["hours-per-week"] >
    upp_bound)]
```

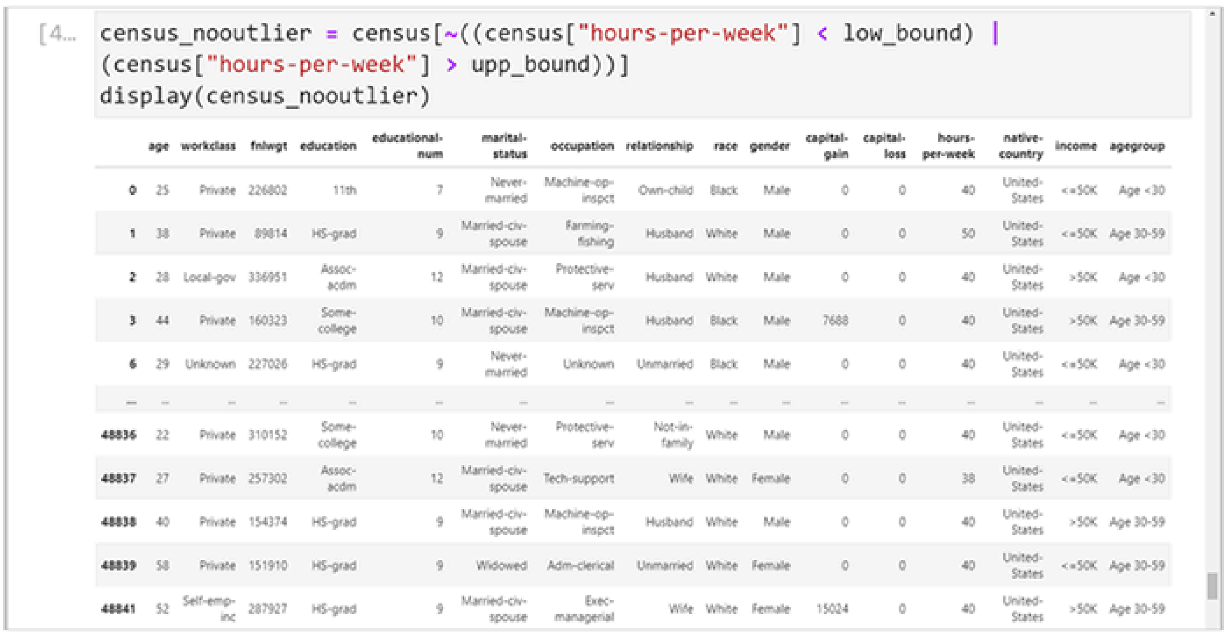
[4…

	age	workclass	fnlwgt	education	educational-num	marital-status	occupation	relationship	race	gender	capital-gain	capital-loss	hours-per-week	native-country	income	agegroup
4	18	Unknown	103497	Some-college	10	Never-married	Unknown	Own-child	White	Female	0	0	30	United-States	<=50K	Age <30
5	34	Private	198693	10th	6	Never-married	Other-service	Not-in-family	White	Male	0	0	30	United-States	<=50K	Age 30-59
7	63	Self-emp-not-inc	104626	Prof-school	15	Married-civ-spouse	Prof-specialty	Husband	White	Male	3103	0	32	United-States	>50K	Age 60+
9	55	Private	104996	7th-8th	4	Married-civ-spouse	Craft-repair	Husband	White	Male	0	0	10	United-States	<=50K	Age 30-59
16	20	State-gov	444554	Some-college	10	Never-married	Other-service	Own-child	White	Male	0	0	25	United-States	<=50K	Age <30
…	…	…	…	…	…	…	…	…	…	…	…	…	…	…	…	…
48825	31	Private	199655	Masters	14	Divorced	Other-service	Not-in-family	Other	Female	0	0	30	United-States	<=50K	Age 30-59
48826	39	Local-gov	111499	Assoc-acdm	12	Married-civ-spouse	Adm-clerical	Wife	White	Female	0	0	20	United-States	>50K	Age 30-59
48829	65	Self-emp-not-inc	99359	Prof-school	15	Never-married	Prof-specialty	Not-in-family	White	Male	1086	0	60	United-States	<=50K	Age 60+
48834	32	Private	116138	Masters	14	Never-married	Tech-support	Not-in-family	Asian-Pac-Islander	Male	0	0	11	Taiwan	<=50K	Age 30-59
48840	22	Private	201490	HS-grad	9	Never-married	Adm-clerical	Own-child	White	Male	0	0	20	United-States	<=50K	Age <30

Figure 4.41 Selected Outlier Observations from a DataFrame

Before dropping those outlier observations from the DataFrame, we shall actually study them more carefully. For instance, the observation with row index 48829 works for 60 hours per week, which is much higher than the third quartile of the data. But according to the variable `workclass`, he is self-employed. From this perspective, his average weekly working hours seem sensible. Hence, this observation could be useful for further analyses.

Nevertheless, in order to show how the syntax works, we will still drop all the outlier observations from `census` that fulfil the above criteria by the following program.



```
[4… census_nooutlier = census[~((census["hours-per-week"] < low_bound) |
    (census["hours-per-week"] > upp_bound))]
    display(census_nooutlier)
```

	age	workclass	fnlwgt	education	educational-num	marital-status	occupation	relationship	race	gender	capital-gain	capital-loss	hours-per-week	native-country	income	agegroup
0	25	Private	226802	11th	7	Never-married	Machine-op-inspct	Own-child	Black	Male	0	0	40	United-States	<=50K	Age <30
1	38	Private	89814	HS-grad	9	Married-civ-spouse	Farming-fishing	Husband	White	Male	0	0	50	United-States	<=50K	Age 30-59
2	28	Local-gov	336951	Assoc-acdm	12	Married-civ-spouse	Protective-serv	Husband	White	Male	0	0	40	United-States	>50K	Age <30
3	44	Private	160323	Some-college	10	Married-civ-spouse	Machine-op-inspct	Husband	Black	Male	7688	0	40	United-States	>50K	Age 30-59
6	29	Unknown	227026	HS-grad	9	Never-married	Unknown	Unmarried	Black	Male	0	0	40	United-States	<=50K	Age <30
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
48836	22	Private	310152	Some-college	10	Never-married	Protective-serv	Not-in-family	White	Male	0	0	40	United-States	<=50K	Age <30
48837	27	Private	257302	Assoc-acdm	12	Married-civ-spouse	Tech-support	Wife	White	Female	0	0	38	United-States	<=50K	Age <30
48838	40	Private	154374	HS-grad	9	Married-civ-spouse	Machine-op-inspct	Husband	White	Male	0	0	40	United-States	>50K	Age 30-59
48839	58	Private	151910	HS-grad	9	Widowed	Adm-clerical	Unmarried	White	Female	0	0	40	United-States	<=50K	Age 30-59
48841	52	Self-emp-inc	287927	HS-grad	9	Married-civ-spouse	Exec-managerial	Wife	White	Female	15024	0	40	United-States	>50K	Age 30-59

Figure 4.42 Selected Non-Outlier Observations from a DataFrame

The construction of the syntax for selecting those non-outliers is rather straightforward. All we need to do is to place a bitwise not operator "~" in front of the entire selection condition for the outliers that is now wrapped up in a round bracket.

Read

Refer to the link below for more details and examples on the `.quantile()` method of the pandas package:

https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.quantile.html

Chapter 5: Data Modification

Lesson Recording

Data Modification in pandas

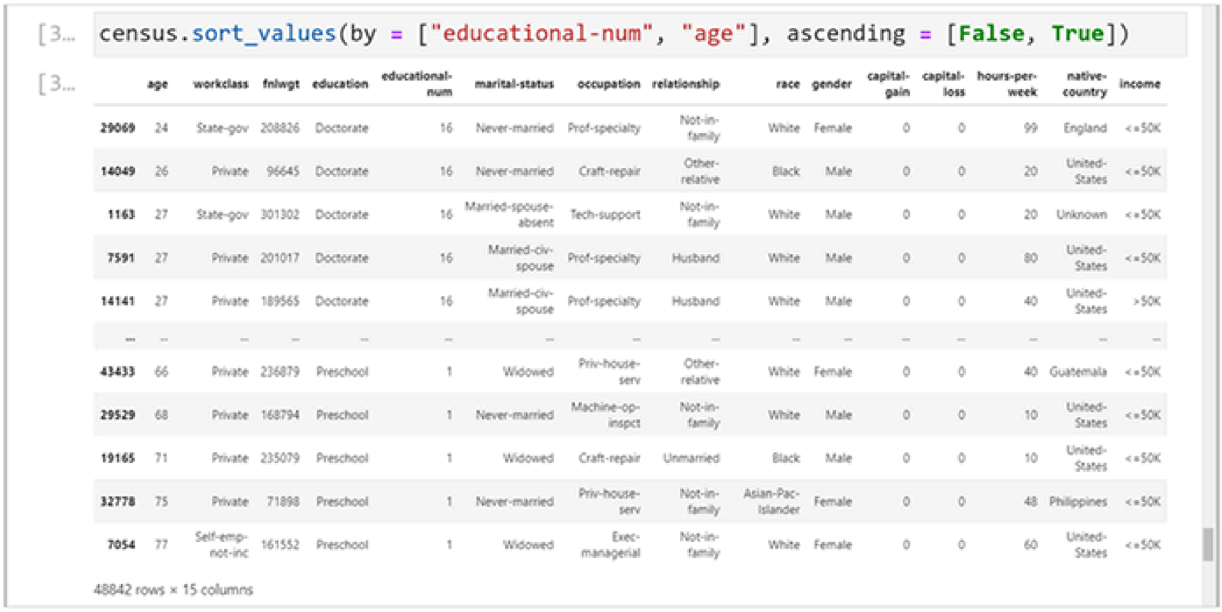
5.1 Sorting Data

The order of the observations in a DataFrame is usually rather arbitrary and random. It can be a result of the sequence in which the data were collected or recorded, or in which they were merged. Sometimes, we may want to sort the data according to values of some variables for better understanding. For instance, we may want to sort an employee dataset by the rank of the employees in the organisation. In Python, the `.sort_values()` method from the pandas package helps us to rearrange the order of the rows in a DataFrame.

```
DataFrame_name.sort_values(by = [List_of_var_names],
 ascending)
```

We can provide a list of variable names to the parameter based on which the DataFrame will be sorted. The sorting hierarchy among these variables drops with the increasing index in the list. If we set the parameter `ascending` to `True`, the values of the variables given in the parameter will be sorted in the ascending order, and they will be sorted in the descending order if it is `False`.

Example (Cont'd): Suppose we would like to study the relationship between the individuals' income and their educational level, which is represented by the numeric variable `educational-num` in the DataFrame `census`, as well as their age. The value in `educational-num` increases with the educational level of the individual. For this purpose, we will sort the DataFrame `census` first by `educational-num` in the descending order and then by `age` in the ascending order. That is, we will see observations with the highest educational level first, and observations with the lowest educational level will appear at the end. And in each educational level, we will first find the youngest individual, and the oldest individual will be put as last in the group.



```
[3… census.sort_values(by = ["educational-num", "age"], ascending = [False, True])
```

[3…

	age	workclass	fnlwgt	education	educational-num	marital-status	occupation	relationship	race	gender	capital-gain	capital-loss	hours-per-week	native-country	income
29069	24	State-gov	208826	Doctorate	16	Never-married	Prof-specialty	Not-in-family	White	Female	0	0	99	England	<=50K
14049	26	Private	96645	Doctorate	16	Never-married	Craft-repair	Other-relative	Black	Male	0	0	20	United-States	<=50K
1163	27	State-gov	301302	Doctorate	16	Married-spouse-absent	Tech-support	Not-in-family	White	Male	0	0	20	Unknown	<=50K
7591	27	Private	201017	Doctorate	16	Married-civ-spouse	Prof-specialty	Husband	White	Male	0	0	80	United-States	<=50K
14141	27	Private	189565	Doctorate	16	Married-civ-spouse	Prof-specialty	Husband	White	Male	0	0	40	United-States	>50K
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
43433	66	Private	236879	Preschool	1	Widowed	Priv-house-serv	Other-relative	White	Female	0	0	40	Guatemala	<=50K
29529	68	Private	168794	Preschool	1	Never-married	Machine-op-inspct	Not-in-family	White	Male	0	0	10	United-States	<=50K
19165	71	Private	235079	Preschool	1	Widowed	Craft-repair	Unmarried	Black	Male	0	0	10	United-States	<=50K
32778	75	Private	71898	Preschool	1	Never-married	Priv-house-serv	Not-in-family	Asian-Pac-Islander	Female	0	0	48	Philippines	<=50K
7054	77	Self-emp-not-inc	161552	Preschool	1	Widowed	Exec-managerial	Not-in-family	White	Female	0	0	60	United-States	<=50K

48842 rows × 15 columns

Figure 4.43 Sorting a DataFrames by Two Variables

In the above syntax, we used two variables for the sorting process. As mentioned, the sorting hierarchy decreases with the index of the variable name in the list. That is, the DataFrame will be first sorted by `education-num`, followed by `age`. As a result, we also need a list of two Boolean values to instruct Python on how each of the variables should be sorted. Here, we ask Python to sort `education-num` in the descending order (`False`) and then `age` in the ascending order (`True`).

Read

Refer to the link below for more details and examples on the `sort_values()` function of the pandas package:

https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.sort_values.html

5.2 Discretisation

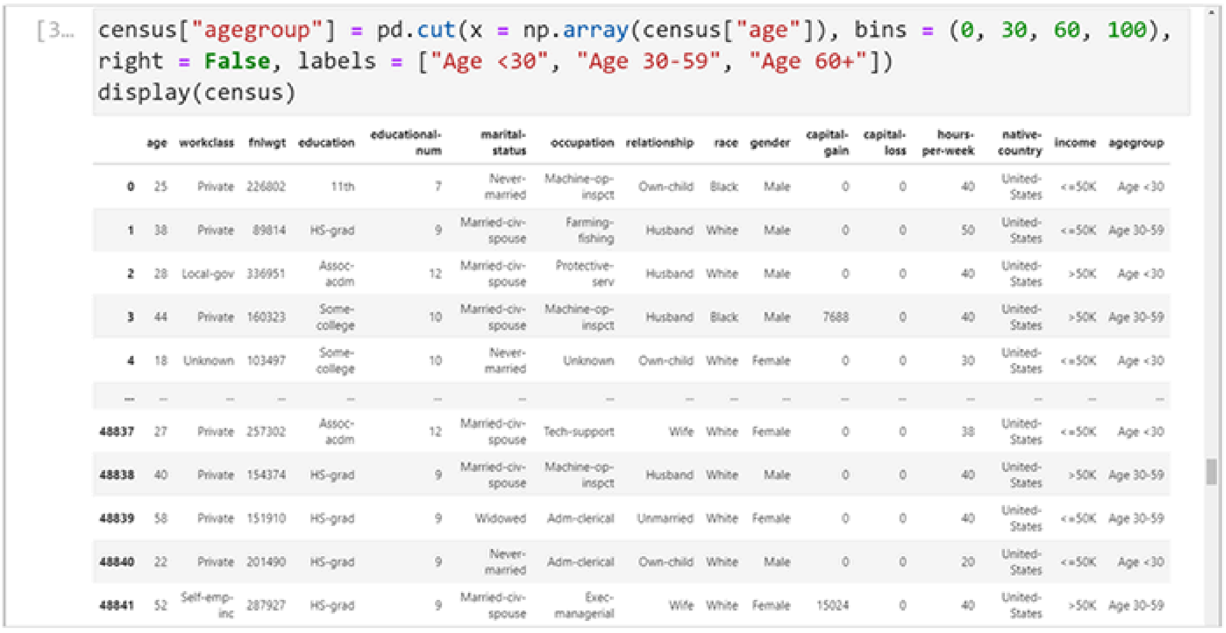
Sometimes, we need to bin continuous variables into discrete intervals. Through discretisation, the variable could be easier to understand or becomes compatible to some specific analytics models such as decision trees. In the past, we may need to write lengthy programs with various number of `if`-conditions for this purpose. In Python, we can use the `cut()` function from the pandas package to discretise continuous variables.

```
DataFrame_name["column"] = pd.cut(x = array, bins, right,
 labels, include_lowest, ordered)
```

Note that `cut()` is a function and not a method to be applied on the DataFrame directly. The object left from the equal sign can be any object including a new or an existing column of a DataFrame.

The data to be discretised should be converted to a one-dimensional NumPy array and assigned to the parameter `x`. With the parameter `bins` we can specify the number of equal-width bins for the discretisation of the array. But we can also define the bin edges in a numeric tuple or numeric list instead. The parameter `right` indicates whether the bins should include the rightmost edge or not. If it is `False`, the leftmost edge will be included instead. Note that one bin edge must be excluded in the discretisation in order not to have overlapping edges. Since the default value here is `True`, Python usually includes the highest value in the corresponding bin. Therefore, the left edge of the first bin is not included as well by default. By assigning `True` to the parameter `include_lowest` we can instruct Python to include the left edge of the first bin. We can also name the bins by assigning a list of strings to the parameter `labels`. And they can be ordered if we assign `True`, the default value here, to the parameter `ordered`.

Example (Cont'd): In Figure 4.9, we use the age groups categorised by the variable `age` as the row index of the DataFrame `census`. There, we wrote a lengthy program to discretise the age into three bins: "Age <30", "Age 30-59", "Age 60+". Here, we can apply the `cut()` function for the same task. The resulting array will then be assigned to a new variable named "agegroup" in `census`.



```
[3...  census["agegroup"] = pd.cut(x = np.array(census["age"]), bins = (0, 30, 60, 100),
       right = False, labels = ["Age <30", "Age 30-59", "Age 60+"])
       display(census)
```

	age	workclass	fnlwgt	education	educational-num	marital-status	occupation	relationship	race	gender	capital-gain	capital-loss	hours-per-week	native-country	income	agegroup
0	25	Private	226802	11th	7	Never-married	Machine-op-inspct	Own-child	Black	Male	0	0	40	United-States	<=50K	Age <30
1	38	Private	89814	HS-grad	9	Married-civ-spouse	Farming-fishing	Husband	White	Male	0	0	50	United-States	<=50K	Age 30-59
2	28	Local-gov	336951	Assoc-acdm	12	Married-civ-spouse	Protective-serv	Husband	White	Male	0	0	40	United-States	>50K	Age <30
3	44	Private	160323	Some-college	10	Married-civ-spouse	Machine-op-inspct	Husband	Black	Male	7688	0	40	United-States	>50K	Age 30-59
4	18	Unknown	103497	Some-college	10	Never-married	Unknown	Own-child	White	Female	0	0	30	United-States	<=50K	Age <30
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
48837	27	Private	257302	Assoc-acdm	12	Married-civ-spouse	Tech-support	Wife	White	Female	0	0	38	United-States	<=50K	Age <30
48838	40	Private	154374	HS-grad	9	Married-civ-spouse	Machine-op-inspct	Husband	White	Male	0	0	40	United-States	>50K	Age 30-59
48839	58	Private	151910	HS-grad	9	Widowed	Adm-clerical	Unmarried	White	Female	0	0	40	United-States	<=50K	Age 30-59
48840	22	Private	201490	HS-grad	9	Never-married	Adm-clerical	Own-child	White	Male	0	0	20	United-States	<=50K	Age <30
48841	52	Self-emp-inc	287927	HS-grad	9	Married-civ-spouse	Exec-managerial	Wife	White	Female	15024	0	40	United-States	>50K	Age 30-59

Figure 4.44 Discretising a Numeric Variable into Bins

In the above syntax, we first convert the variable `age` from the `census` DataFrame into a NumPy array and assign it to the parameter `x`. Then we specify in `bins` the edges of the three bins in a tuple `(0, 30, 60, 100)`, and the right edges should not be included here as we set `right = False`. These settings enable us to use 30 instead of 29, 60 instead of 59 as bin edges. Since there is no `include_highest` parameter for the `cut()` function, we must set the rightmost edge of the last group higher than the maximum age in our DataFrame. In the final step, we use the same labels as in Figure 4.9 for our bins and specify them as a list for the parameter `labels`.

Read

Refer to the link below for more details and examples on the `.cut()` method of the pandas package:

https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.cut.html

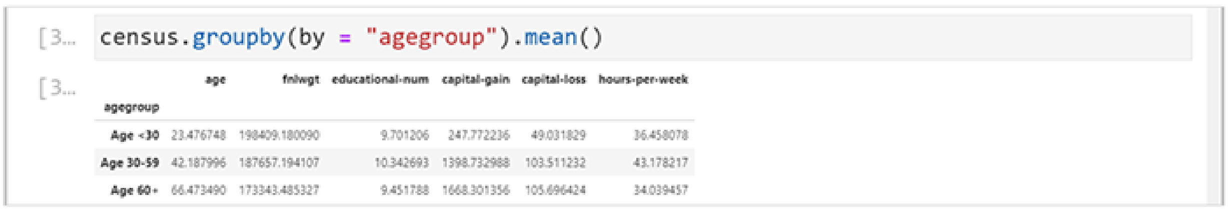
5.3 Grouping Data

In data analytics, we often need to group the data by one or more variables and compute the aggregated statistics of some other variables for each group. To group a DataFrame by some variables in Python, we can use the `.groupby()` method of the pandas package.

```
DataFrame_name.groupby(by = [List_of_Labels]).anymethod()
```

With the parameter `by` we can specify a list of column labels, or variable names, based on which the grouping should be conducted. These variables must be categorical so that the number of groups is finite and limited. Attached to the `.groupby()` method can be any method that we would like to apply on the grouped data. The list of such functions or methods can be found in Table 3.1 of Study Unit 3 since the NumPy functions or methods are also applicable to pandas DataFrames.

Example (Cont'd): Suppose we would like to compute the mean of the number of working hours, capital gain and capital loss as well as the age for each age group created in Figure 4.44.

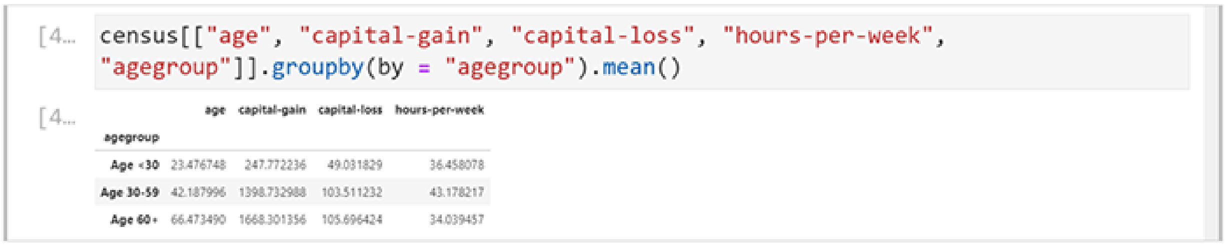


```
[3... census.groupby(by = "agegroup").mean()
```

```
[3...            age        fnlwgt  educational-num  capital-gain  capital-loss  hours-per-week
agegroup
  Age <30  23.476748  198409.180090        9.701206    247.772236     49.031829       36.458078
Age 30-59  42.187996  187657.194107       10.342693   1398.732988    103.511232       43.178217
  Age 60+  66.473490  173343.485327        9.451788   1668.301356    105.696424       34.039457
```

Figure 4.45 Computing the Mean of All Numeric Columns for Grouped Data

The `.mean()` method of pandas selects all columns of type integer or float and compute their means in each age group. As a result, we obtain the group means of `fnlwgt` and `educational-num` as well. To select only the relevant variables for the group mean calculation, we can subset the census DataFrame first in the above syntax.



```
[4... census[["age", "capital-gain", "capital-loss", "hours-per-week",
      "agegroup"]].groupby(by = "agegroup").mean()
```

```
[4...            age  capital-gain  capital-loss  hours-per-week
agegroup
  Age <30  23.476748    247.772236     49.031829       36.458078
Age 30-59  42.187996   1398.732988    103.511232       43.178217
  Age 60+  66.473490   1668.301356    105.696424       34.039457
```

Figure 4.46 Computing the Mean of Selected Columns for Grouped Data

As a result, the average age of the youngest group is 23.4, while those between 39 and 59 is 42.2 and those who are 60 and older is 66.5. Individuals between the age of 30 and 59 have to work averagely over 43 hours per week while the average numbers of the youngsters and seniors are 36 and 34 hours a week, respectively. Furthermore, seniors at the age of 60 or above have on average the highest capital gain and loss in comparison to the other two groups.

Read

Refer to the link below for more details and examples on the functions `groupby()` of the pandas package:

https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.groupby.html

5.4 Transformation, Standardisation, Normalisation

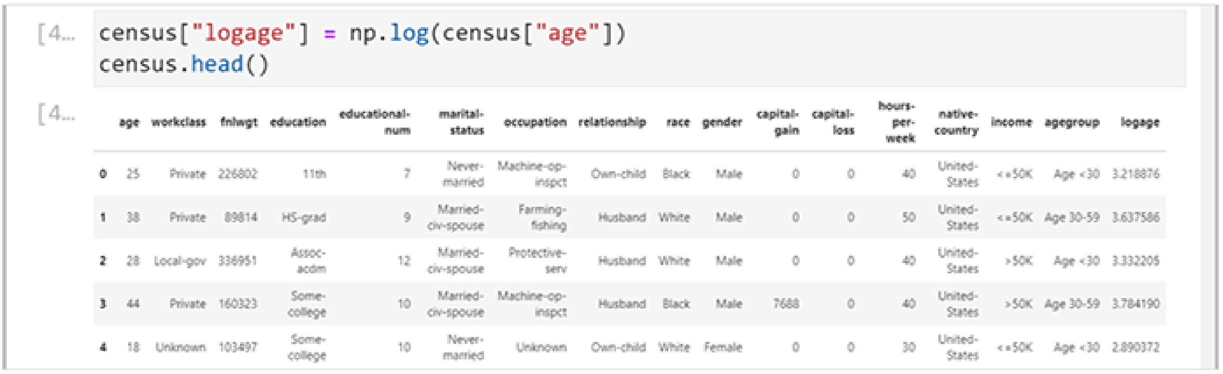
In data analytics, we may need to transform the values of a variable due to various reasons. For instance, we can use the log-transformation to stabilise the variance of a variable, or we need to standardise or normalise variables for customer segmentation analysis when they are measured at different scales and do not contribute equally to the analysis. In Python, we can use various functions to transform, standardise or normalise variables.

The log-transformation of a numeric variable is rather straightforward. Since the `log()` function is not available in the pandas package, we need to take it from the NumPy package.

```
DataFrame_name["new_var"] =
 np.log(DataFrame_name["var_name"])
```

It is often useful not to replace the values in the original variable by transformed values since we may still need the original one for other purposes later. As a result, we shall save the transformed values as a new variable in the same DataFrame.

Example (Cont'd): Suppose we would like to transform `age` by the natural logarithm for further analyses such as a Gamma regression.



```
[4... census["logage"] = np.log(census["age"])
      census.head()
```

[4...

	age	workclass	fnlwgt	education	educational-num	marital-status	occupation	relationship	race	gender	capital-gain	capital-loss	hours-per-week	native-country	income	agegroup	logage
0	25	Private	226802	11th	7	Never-married	Machine-op-inspct	Own-child	Black	Male	0	0	40	United-States	<=50K	Age <30	3.218876
1	38	Private	89814	HS-grad	9	Married-civ-spouse	Farming-fishing	Husband	White	Male	0	0	50	United-States	<=50K	Age 30-59	3.637586
2	28	Local-gov	336951	Assoc-acdm	12	Married-civ-spouse	Protective-serv	Husband	White	Male	0	0	40	United-States	>50K	Age <30	3.332205
3	44	Private	160323	Some-college	10	Married-civ-spouse	Machine-op-inspct	Husband	Black	Male	7688	0	40	United-States	>50K	Age 30-59	3.784190
4	18	Unknown	103497	Some-college	10	Never-married	Unknown	Own-child	White	Female	0	0	30	United-States	<=50K	Age <30	2.890372

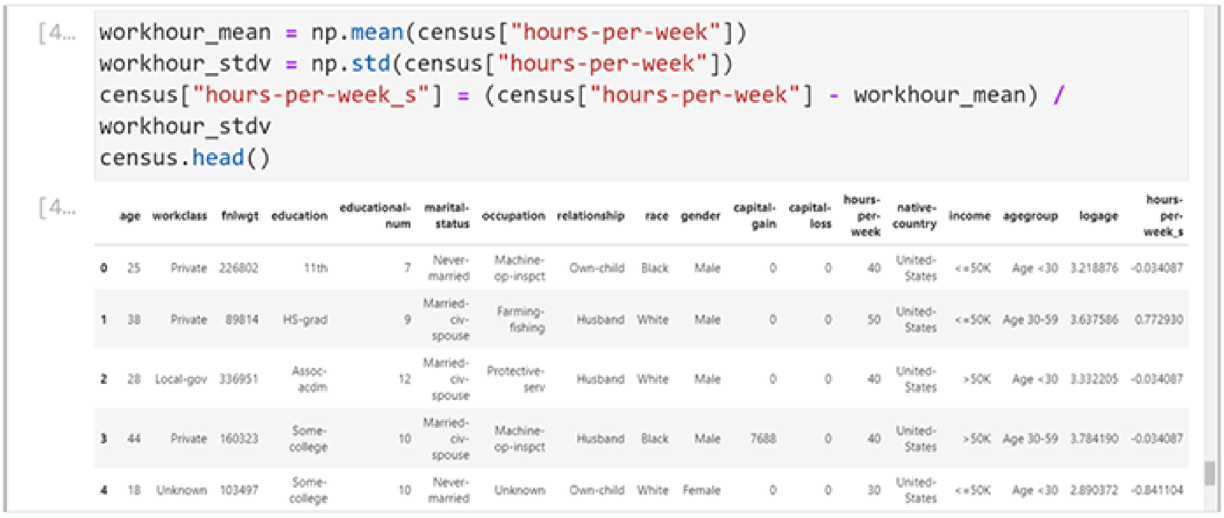
Figure 4.47 Log-Transformation of a Numeric Variable

In Python, the standardisation function can be found in the "scikit-learn" package which we will introduce in Study Unit 5. Here, we use the most traditional way to standardise a variable by finding its mean and standard deviation first, and the transformation will be then conducted by a formula.

```
var_mean = np.mean(DF["var_name"])
var_std = np.std(DF["var_name"])
DF["std_var"] = (DF["var_name"] – var_mean) / var_std
```

We can certainly write all the three lines into a single one without assigning the variable mean and variable standard deviation to different variables first. The advantage of splitting such a long syntax into three short ones is the readability of the code and convenience in debugging.

Example (Cont'd): Suppose we would like to standardise `hours-per-week` for further analyses.



```
[4... workhour_mean = np.mean(census["hours-per-week"])
      workhour_stdv = np.std(census["hours-per-week"])
      census["hours-per-week_s"] = (census["hours-per-week"] - workhour_mean) /
      workhour_stdv
      census.head()
```

[4...

	age	workclass	fnlwgt	education	educational-num	marital-status	occupation	relationship	race	gender	capital-gain	capital-loss	hours-per-week	native-country	income	agegroup	logage	hours-per-week_s
0	25	Private	226802	11th	7	Never-married	Machine-op-inspct	Own-child	Black	Male	0	0	40	United-States	<=50K	Age <30	3.218876	-0.034087
1	38	Private	89814	HS-grad	9	Married-civ-spouse	Farming-fishing	Husband	White	Male	0	0	50	United-States	<=50K	Age 30-59	3.637586	0.772930
2	28	Local-gov	336951	Assoc-acdm	12	Married-civ-spouse	Protective-serv	Husband	White	Male	0	0	40	United-States	>50K	Age <30	3.332205	-0.034087
3	44	Private	160323	Some-college	10	Married-civ-spouse	Machine-op-inspct	Husband	Black	Male	7688	0	40	United-States	>50K	Age 30-59	3.784190	-0.034087
4	18	Unknown	103497	Some-college	10	Never-married	Unknown	Own-child	White	Female	0	0	30	United-States	<=50K	Age <30	2.890372	-0.841104

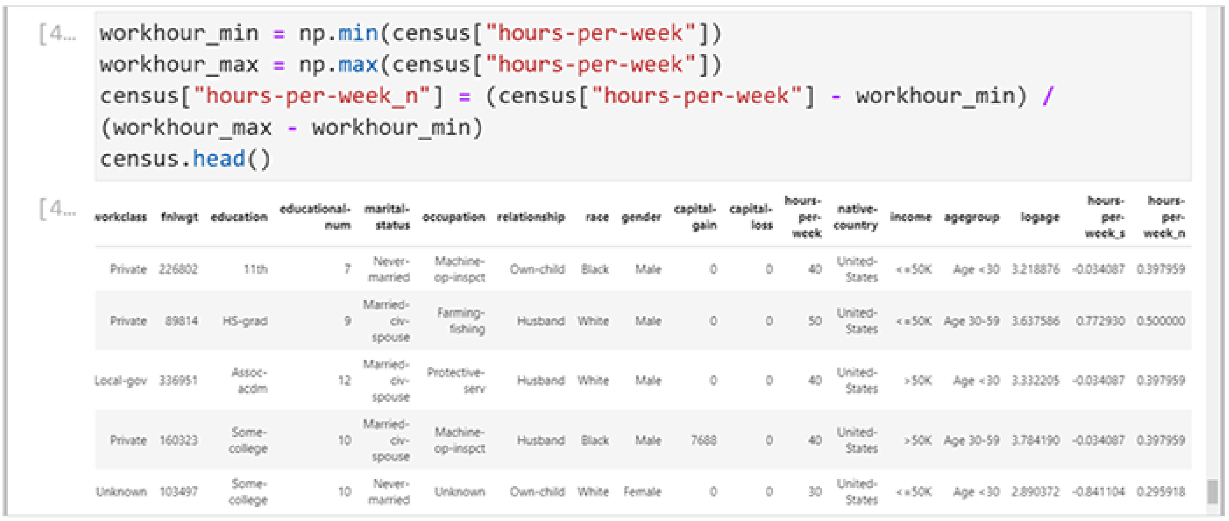
Figure 4.48 Standardisation of a Numeric Variable

Normalisation is another transformation method to scale down a variable. While there are no theoretical upper and lower bounds for standardised variables, the values of a normalised variable can only be in the interval [0, 1]. Same as the standardisation function, the normalisation function in Python can also be found in the "scikit-learn" package. Here, we use the most traditional way to normalise a variable.

```
var_min = np.min(DF["var_name"])
var_max = np.max(DF["var_name"])
DF["norm_var"] = (DF["var_name"] – var_min) / (var_max –
 var_min)
```

Same as the syntax for standardisation, we need to find the minimum and maximum of the target variable first and then transform the variable by a formula.

Example (Cont'd): We will now normalise `hours-per-week` for further analyses.



```
[4… workhour_min = np.min(census["hours-per-week"])
    workhour_max = np.max(census["hours-per-week"])
    census["hours-per-week_n"] = (census["hours-per-week"] - workhour_min) /
    (workhour_max - workhour_min)
    census.head()
```

[4…

workclass	fnlwgt	education	educational-num	marital-status	occupation	relationship	race	gender	capital-gain	capital-loss	hours-per-week	native-country	income	agegroup	logage	hours-per-week_s	hours-per-week_n
Private	226802	11th	7	Never-married	Machine-op-inspct	Own-child	Black	Male	0	0	40	United-States	<=50K	Age <30	3.218876	-0.034087	0.397959
Private	89814	HS-grad	9	Married-civ-spouse	Farming-fishing	Husband	White	Male	0	0	50	United-States	<=50K	Age 30-59	3.637586	0.772930	0.500000
Local-gov	336951	Assoc-acdm	12	Married-civ-spouse	Protective-serv	Husband	White	Male	0	0	40	United-States	>50K	Age <30	3.332205	-0.034087	0.397959
Private	160323	Some-college	10	Married-civ-spouse	Machine-op-inspct	Husband	Black	Male	7688	0	40	United-States	>50K	Age 30-59	3.784190	-0.034087	0.397959
Unknown	103497	Some-college	10	Never-married	Unknown	Own-child	White	Female	0	0	30	United-States	<=50K	Age <30	2.890372	-0.841104	0.295918

Figure 4.49 Normalisation of a Numeric Variable

From Figure 4.49, we can clearly see that the values of the normalised variable `hours-per-week_n` are entirely non-negative while the standardised variable `hours-per-week_s` contains both positive and negative values.

Summary

In this unit, we have seen how Python can be used to manipulate, clean, and query data using the pandas package. Querying the DataFrame structures can be done in different ways, such as using the `.iloc` or `.loc` attributes for row-based querying or using the square brackets on the object itself for column-based querying. We also saw that one can query the DataFrame through Boolean masking. Furthermore, we also came across situations where we had to use the `.append()` method and the `concat()` function to merge multiple DataFrames with different shapes into one. We then explored how to detect and replace missing values as well as outliers in a DataFrame. We also talked about modifying DataFrames for further analyses such as sorting and grouping data, discretising numeric variables to finite number of categories or bins. While pandas offers specific methods or functions such as `.sort_values()`, `.groupby()`, or `cut()` for these types of data modification, we need to construct our own syntax by combining various NumPy functions to log-transform, to standardise, or to normalise variables if transformation is required for a variable by the analytics methods.

Formative Assessment

1. What is not a function/method to display a DataFrame?

   a. `print()`

   b. `.head()`

   c. `show()`

   d. `display()`

2. Which of the following values can be used with `.iloc` to select rows from a DataFrame?

   a. `["var_label1", "var_label2"]`

   b. `[-4:-1]`

   c. `["0", "1", "2"]`

   d. `DataFrame_name["var_label1" == "good"]`

3. What is the resulting DataFrame according to the following program?

```
df = pd.DataFrame([[1, 2]], columns = list('AB'))
df2 = pd.DataFrame([[3, 4]], columns = list('AB'))
df.append(df2)
```

   a.

A	B
1	2
3	4

   b.

A	B
3	4

1	2

c.

A	B
1	3
2	4

d.

A	B
3	1
4	2

4. Which columns does `dfnew` contain according to the following program?

```
df = pd.DataFrame([[1, 2]], columns = list('AB'))
df2 = pd.DataFrame([[3, 4]], columns = list('BC'))
dfnew = pd.concat([df, df2], join = 'outer')
```

   a. `A` and `B`
   b. `B` and `C`
   c. Only `B`
   d. `A`, `B`, and `C`

5. Which argument of the `pd.read_csv()` function is used to declare specific strings as missing values?
   a. `na_values`
   b. `na_filter`
   c. `na_drop`
   d. `na_omit`

6. Which method should be used to replace missing values by another value?
   a. `.drop()`
   b. `.dropna()`
   c. `.nareplace()`
   d. `.fillna()`

7. Which variable is the lowest in the sorting hierarchy?

```
df.sort_values(by = ['Z', 'Y', 'X', 'W'])
```

   a. `X`
   b. `W`
   c. `Z`
   d. `Y`

8. What function of the pandas package is used to discretise numeric variables?
   a. `pd.bins()`
   b. `pd.categorize()`
   c. `pd.cut()`
   d. `pd.split()`

9. What is not recommended when applying the `.groupby()` method on a DataFrame?
   a. The grouping variable should be categorical.
   b. The grouping variable should be discretised before.
   c. The grouping variable should be of type float.
   d. The values in the grouping variable can be identical to the row indices.

10. Which functions of the NumPy package do we need to normalise a variable?
   a. `np.min()` and `np.max()`
   b. `np.mean()` and `np.std()`

c. `np.quantile()` and `np.range()`

d. `np.cov()` and `np.corr()`

Solutions or Suggested Answers

Formative Assessment

1. What is not a function/method to display a DataFrame?

   a. `print()`

   Incorrect. We can use the `print()` function to print out a DataFrame.

   b. `.head()`

   Incorrect. We can use the `.head()` method to print out the first five rows of a DataFrame.

   c. `show()`

   Correct. `show()` is a function of matplotlib which is used to show the created graphs.

   d. `display()`

   Incorrect. We can use the `display()` function to print out the first five and the last five rows of a DataFrame.

2. Which of the following values can be used with `.iloc` to select rows from a DataFrame?

   a. `["var_label1", "var_label2"]`

   Incorrect. We can only apply `.iloc` on row positions.

   b. `[-4:-1]`

   Correct. -4:-1 are clearly indices representing the row positions to be selected.

   c. `["0", "1", "2"]`

Incorrect. The values in the index operator for `.iloc` must be numeric and not label strings.

d. `DataFrame_name["var_label1" == "good"]`

Incorrect. The arguments in the index operator for `.iloc` must be numeric values and not Boolean expressions.

3. What is the resulting DataFrame according to the following program?

```
df = pd.DataFrame([[1, 2]], columns = list('AB'))
df2 = pd.DataFrame([[3, 4]], columns = list('AB'))
df.append(df2)
```

a.

```
A  B

1  2

3  4
```

Correct. The `.append()` method merges two DataFrames by row.

b.

```
A  B

3  4

1  2
```

Incorrect. The `.append()` method appends `df2` to `df1` and not vice versa.

c.

```
A  B

1  3
```

```
2   4
```

Incorrect. The `.append()` method does not merge DataFrames by column. Besides, `df` and `df2` are row DataFrames and not column DataFrames.

d.

```
A   B

3   1

4   2
```

Incorrect. The `.append()` method does not merge DataFrames by column. Besides, `df` and `df2` are row DataFrames and not column DataFrames. Furthermore, it should be `df2` appended to `df1` and not `df1` to `df2`.

4. Which columns does `dfnew` contain according to the following program?

```
df = pd.DataFrame([[1, 2]], columns = list('AB'))
df2 = pd.DataFrame([[3, 4]], columns = list('BC'))
dfnew = pd.concat([df, df2], join = 'outer')
```

a. `A` and `B`

Incorrect. Since it is an outer join, the resulting DataFrame should contain all available columns across the original DataFrames and not only those from `df`.

b. `B` and `C`

Incorrect. Since it is an outer join, the resulting DataFrame should contain all available columns across the original DataFrames and not only those from `df2`.

c. Only `B`

Incorrect. Since it is an outer join and not inner join, the resulting DataFrame should contain all available columns across the original DataFrames and not only the common ones.

d. `A`, `B`, and `C`

Correct. Since it is an outer join, the resulting DataFrame should contain all available columns across the original DataFrames, which are A, B and C.

5. Which argument of the `pd.read_csv()` function is used to declare specific strings as missing values?

   a. `na_values`

   Correct. We can specify a list of strings representing missing values in the DataFrame with `na_values`.

   b. `na_filter`

   Incorrect. `na_filter` is used to convert white spaces to missing values.

   c. `na_drop`

   Incorrect. There is no `na_drop` parameter for the `pd.read_csv()` function.

   d. `na_omit`

   Incorrect. There is no `na_omit` parameter for the `pd.read_csv()` function.

6. Which method should be used to replace missing values by another value?

   a. `.drop()`

   Incorrect. The `.drop()` method is used to delete an entire row/column from a DataFrame.

   b. `.dropna()`

Incorrect. The `.dropna()` method is used to delete an entire row from a DataFrame if it contains any missing value.

c. `.nareplace()`

Incorrect. There is no method called `.nareplace()` in the pandas package.

d. `.fillna()`

Correct. With the `.fillna()` method, we can replace missing values in a DataFrame by a user-defined value.

7. Which variable is the lowest in the sorting hierarchy?

```
df.sort_values(by = ['Z', 'Y', 'X', 'W'])
```

a. `X`

Incorrect. The sorting hierarchy decreases with the increase of the index in the `by` list. As a result, variable `W` should be the lowest in the hierarchy.

b. `W`

Correct. The sorting hierarchy decreases with the increase of the index in the `by` list. As a result, variable `W` is the lowest in the sorting hierarchy.

c. `Z`

Incorrect. The sorting hierarchy decreases with the increase of the index in the `by` list. As a result, variable `W` should be the lowest in the hierarchy.

d. `Y`

Incorrect. The sorting hierarchy decreases with the increase of the index in the `by` list. As a result, variable `W` should be the lowest in the hierarchy.

8. What function of the pandas package is used to discretise numeric variables?

a. `pd.bins()`

   Incorrect. There is no function called `bins()` in the pandas package.

b. `pd.categorize()`

   Incorrect. There is no function called `categorize()` in the pandas package.

c. `pd.cut()`

   Correct. We can use the `cut()` function to discretise numeric variables into bins.

d. `pd.split()`

   Incorrect. There is no function called `split()` in the pandas package.

9. What is not recommended when applying the `.groupby()` method on a DataFrame?

   a. The grouping variable should be categorical.

      Incorrect. The number of possible values of a grouping variable should be limited. A categorical variable matches this criterion perfectly.

   b. The grouping variable should be discretised before.

      Incorrect. If the grouping variable is the result of a discretised numeric variable, then its number of possible values has been limited, which matches the criterion of a grouping variable perfectly.

   c. The grouping variable should be of type float.

      Correct. This is not recommendable since the number of possible values in a grouping variable should be limited. However, the number of possible values of a float variable is infinite.

   d. The values in the grouping variable can be identical to the row indices.

Incorrect. In the case that the row indices are values of a categorical variable, this categorical variable can perfectly be used as a grouping variable since its number of possible values is limited.

10. Which functions of the NumPy package do we need to normalise a variable?

    a. `np.min()` and `np.max()`

    Correct. We need the minimum and maximum of a variable for the normalisation formula.

    b. `np.mean()` and `np.std()`

    Incorrect. The mean and standard deviation of a variable are used for the standardisation.

    c. `np.quantile()` and `np.range()`

    Incorrect. The quantile and range functions are not required for normalisation.

    d. `np.cov()` and `np.corr()`

    Incorrect. Covariance and correlation are not needed for normalisation.

References

Delve. (1996). *The Adult dataset*. The University of Toronto. http://www.cs.toronto.edu/~delve/data/adult/adultDetail.html

Kaggle. (n.d.). *Adult income dataset*. Kaggle Inc. https://kaggle.com/wenruliu/adult-income-dataset

pandas. (n.d.). *IO tools (text, CSV, HDF5, ...)*. The pandas development team. https://pandas.pydata.org/docs/user_guide/io.html#io-read-csv-table

pandas. (n.d.). *Merge, join, concatenate and compare*. The pandas development team. https://pandas.pydata.org/pandas-docs/stable/user_guide/merging.html

pandas. (n.d.). *pandas.concat*. The pandas development team. https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.concat.html

pandas. (n.d.). *pandas.cut*. The pandas development team. https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.cut.html

pandas. (n.d.). *pandas.DataFrame.any*. The pandas development team. https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.any.html

pandas. (n.d.). *pandas.DataFrame.append*. The pandas development team. https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.append.html

pandas. (n.d.). *pandas.DataFrame.drop*. The pandas development team. https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.drop.html

pandas. (n.d.). *pandas.DataFrame.dropna*. The pandas development team. https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.dropna.html pandas. (n.d.). *pandas.DataFrame.fillna*. The pandas development team. https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.fillna.html

pandas. (n.d.). *pandas.DataFrame.groupby*. The pandas development team. https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.groupby.html

pandas. (n.d.). *pandas.DataFrame.iloc*. The pandas development team. https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.iloc.html

pandas. (n.d.). *pandas.DataFrame.index*. The pandas development team. https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.index.html

pandas. (n.d.). *pandas.DataFrame.isnull*. The pandas development team. https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.isnull.html

pandas. (n.d.). *pandas.DataFrame.loc*. The pandas development team. https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.loc.html

pandas. (n.d.). *pandas.DataFrame.quantile*. The pandas development team. https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.quantile.html

pandas. (n.d.). *pandas.DataFrame.reset_index*. The pandas development team. https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.reset_index.html

pandas. (n.d.). *pandas.DataFrame.set_index*. The pandas development team. https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.set_index.html pandas. (n.d.). *pandas.DataFrame.sort_values*. The pandas development team. https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.sort_values.html

pandas. (n.d.). *pandas.DataFrame.sum*. The pandas development team. https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.sum.html

pandas. (n.d.). *pandas.read_csv*. The pandas development team. https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.read_csv.html

Study Unit 5

Data Analytics in Python

Learning Outcomes

By the end of this unit, you should be able to:

1. Design Python programmes for performing data analytics
2. Analyse data using appropriate tools for data mining

Overview

In this unit, we will discuss the implementation of two analytics techniques in Python: k-means clustering and decision trees. All these functions, modules, and algorithms can be found in the scikit-learn library. The scikit-learn library is a machine learning library written for the Python programming language. It features various modules such as classification, clustering, regression, etc. We will first develop Python programs to prepare the DataFrame for the different requirements of these sub-packages. We will then learn how the two analytics techniques can be carried out by Python programs and how their results can be extracted and presented.

Chapter 1: Introduction to Scikit-Learn

1.1 Installing and Importing Scikit-Learn

Lesson Recording

Introduction to Scikit-Learn

It has become very common to carry out tasks for data analytics, statistical modelling or machine learning in Python recently. And the trend is rising. In fact, many Python packages in these areas have also been developed. One of the most common libraries used for these purposes is scikit-learn, a free machine learning library written for Python. In this study unit, we will write our code with the scikit-learn functions in JupyterLab.

One reason that scikit-learn has become one of the most common machine learning libraries for programming is its broad applicability and functionality. It features various algorithms for classification, regression, clustering, etc. In machine learning, programs are constructed with parameters such that they can "learn" from newly fed data. That is, they can automatically adjust and improve their behaviour according to the new "knowledge". Below is a table of the most common algorithms that are available in scikit-learn.

Table 5.1 Most Common Algorithms Available in scikit-learn

Supervised learning	
Linear Models	Gaussian Processes
Discriminant Analysis	Cross decomposition
Kernel ridge regression	Decision Trees

Support Vector Machines	Isotonic regression
Nearest Neighbours	Neural network models (supervised)
Unsupervised learning	
Gaussian mixture models	Novelty and Outlier Detection
Clustering	Density Estimation
Covariance estimation	Neural network models (unsupervised)

Beside machine learning algorithms, scikit-learn also provides modules for model selection, visualisation, data transformation as well as example datasets. The website https://scikit-learn.org/stable/user_guide.html contains many details of the library.

Same as NumPy, matplotlib and pandas, we can simply use pip, the package installer of Python, to download and install scikit-learn.

```
pip install scikit-learn
```

After installing scikit-learn using pip, we can import it into our program.

```
import sklearn
```

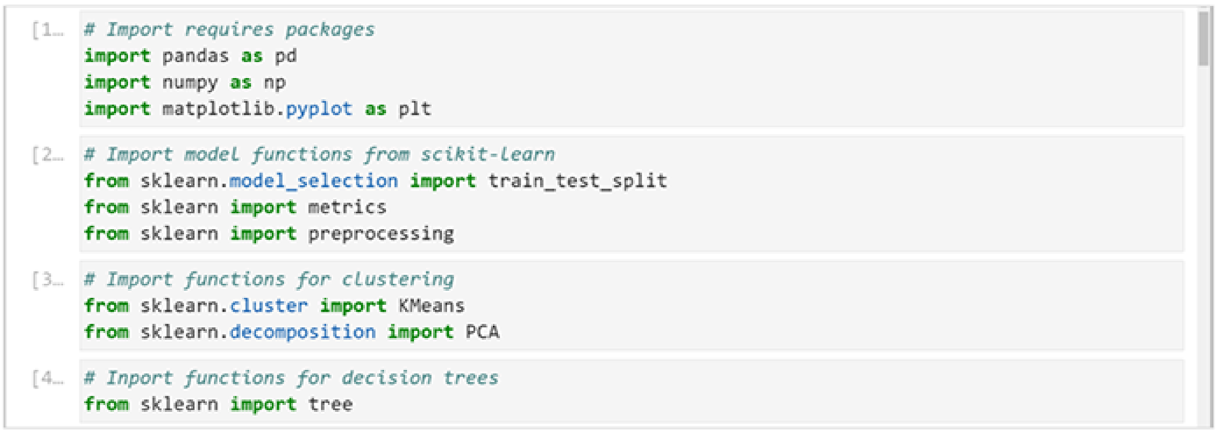
Note that it is `sklearn` and not `scikit-learn` that refers to the scikit-learn library in the Python programs. Nevertheless, since the library is extraordinary extensive, programmers usually do not import the entire library. Instead, the common practice is to load the required algorithm or only its "estimator" object. For instance, if linear regression models are required for the analytics task, we can import the estimator `LinearRegression` from the module `linear_model`.

```
from sklearn.linear_model import LinearRegression
```

Since each module has its own estimators, functions, etc., it is important to refer to the official websites for the correct spelling, including the cases of the names. It is not unusual that we need to load couple of them for a single analytics task. It is therefore important to put sufficient comments in the program to explain the purpose and use of each imported module.

In this study unit, we will demonstrate two scikit-learn algorithms, k-means clustering and decision trees, to show how the library, and Python in general, can be applied in data analytics. But before we can apply these algorithms, we need to prepare the data according to the requirements of each of the algorithms. The preparation process will be discussed in the next section.

Example (Adult Census Data): In this study unit, we will construct programs to estimate the two mentioned machine learning models on the US Adult Census dataset which has been introduced in Study Unit 4. The dataset is a repository of 48,842 entries extracted from the 1994 US Census database to predict whether income would exceed $50,000 per year according to the 14 social-demographic attributes. Before we begin to develop the code using the scikit-learn algorithms, we need to import the corresponding packages or modules first.



```
[1… # Import requires packages
    import pandas as pd
    import numpy as np
    import matplotlib.pyplot as plt

[2… # Import model functions from scikit-learn
    from sklearn.model_selection import train_test_split
    from sklearn import metrics
    from sklearn import preprocessing

[3… # Import functions for clustering
    from sklearn.cluster import KMeans
    from sklearn.decomposition import PCA

[4… # Inport functions for decision trees
    from sklearn import tree
```

Figure 5.1 Importing Modules and Functions from scikit-learn

In the first box, the packages introduced in the two previous study units, pandas, NumPy and matplotlib are imported. We will need them to manage DataFrames, to convert slices of DataFrames to multidimensional arrays, and to construct plots to illustrate and evaluate the model results. In the second box, we import modules from scikit-learn that we need for the pre-processing and transformation of the DataFrames for model constructions. For instance, with the `train_test_split` function, we can instruct Python to split arrays into random training and testing subsets for evaluating the estimator performance. Furthermore, the module `metrics` includes functions to compute metrics and distances for the evaluation of classification performance. The functions in the `preprocessing` module such as scaling, centring, normalisation, etc. are used to prepare DataFrames for the scikit-learn algorithms. In the last two boxes, we import the modules of k-means clustering (`KMeans`) and decision trees (`tree`). In addition to the KMeans module, we also import the PCA module from the decomposition sub-package for dimension reduction, which will be helpful to plot multivariate data as a two-dimensional chart.

Read

Refer to the link below for more details on the installation of the scikit-learn package:

https://scikit-learn.org/stable/install.html

Refer to the link below for more details and examples on the `cluster` module of the scikit-learn package for applying K-Means clustering:

https://scikit-learn.org/stable/modules/clustering.html

Refer to the link below for more details and examples on the `tree` module of the scikit-learn package for constructing decision trees:

https://scikit-learn.org/stable/modules/tree.html

Refer to the link below for more details and examples on the `train_test_split` module of the scikit-learn package for splitting arrays into random training and testing subsets:

https://scikit-learn.org/stable/modules/generated/
sklearn.model_selection.train_test_split.html

Refer to the link below for more details and examples on the `metric` module of the scikit-learn package for metrics and distance computations:

https://scikit-learn.org/stable/modules/classes.html?highlight=metrics#module-sklearn.metrics

Refer to the link below for more details and examples on the `preprocessing` module of the scikit-learn package for data preparation techniques such as scaling, centring, normalisation, etc.:

https://scikit-learn.org/stable/modules/classes.html?highlight=preprocessing#module-sklearn.preprocessing

1.2 Data Preparation for Analytics Algorithms

Lesson Recording

Data Preparation for Analytics Algorithms of scikit-learn (1/2)

Data Preparation for Analytics Algorithms of scikit-learn (2/2)

In Figure 5.1, various modules for data preparation have been imported. At the same time, we have also imported packages that we have already worked with in the previous study units: NumPy, pandas, matplotlib. In fact, the scikit-learn algorithms work hand-in-hand with these packages. In this section, we will need to combine NumPy, pandas and scikit-learn to prepare datasets to meet the requirements of every scikit-learn module.

1.2.1 Missing Values

One of the first steps in data preparation is to check on and to deal with missing values in the dataset. In Chapter 4 of Study Unit 4, we have discussed how to specify, identify, and modify observations with missing values. To recall some details, we can define specific strings in the dataset as missing values during the reading process and, at the same time, we instruct Python to treat white strings as missing values if necessary.

```
DataFrame_name = pd.read_csv("csv_file_name.csv", na_values
                             = "na_string", na_filer
                             = True/False)
```

After creating a pandas DataFrame, we have to decide on the appropriate measure to deal with the missing values in it. One way is to remove those rows with missing values in any of the columns from the DataFrame completely.

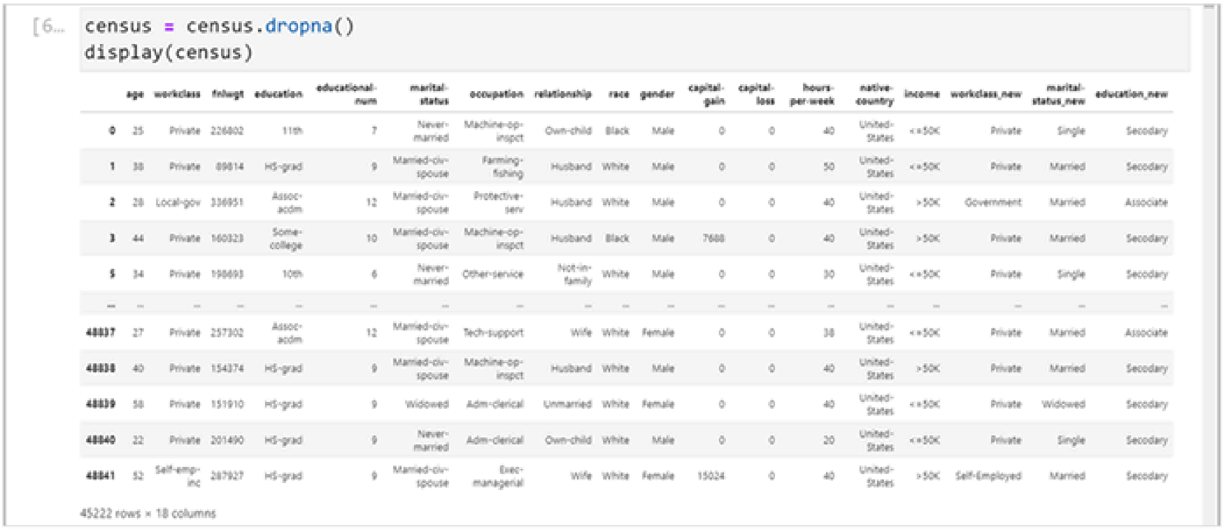
```
DataFrame_name.dropna(axis = 0, how = "any"/"all")
```

Another possibility is to replace them by specific values. Here, we can choose to apply the replacement on missing values of the entire DataFrame or just one specific column.

```
DataFrame_name.fillna(value = repl_value)
DataFrame_name["column_label"].fillna(value = repl_value)
```

The advantage of replacing the missing values in all columns is certainly the convenience in creating the corresponding code. Nevertheless, it is not unusual that a DataFrame contains various types of variables. In this case, replacing all missing values by a single value may be undesirable or impossible. The replacement values should be chosen according to the characteristics and requirements of each variable.

Example (Cont'd): After studying the missing data in the US Adult Census dataset, which occur solely in the variables `workclass`, `occupation`, and `native-country`, it seems appropriate to remove all these rows from the DataFrame.



```
[6...  census = census.dropna()
       display(census)
```

	age	workclass	fnlwgt	education	educational-num	marital-status	occupation	relationship	race	gender	capital-gain	capital-loss	hours-per-week	native-country	income	workclass_new	marital-status_new	education_new
0	25	Private	226802	11th	7	Never-married	Machine-op-inspct	Own-child	Black	Male	0	0	40	United-States	<=50K	Private	Single	Secodary
1	38	Private	89814	HS-grad	9	Married-civ-spouse	Farming-fishing	Husband	White	Male	0	0	50	United-States	<=50K	Private	Married	Secodary
2	28	Local-gov	336951	Assoc-acdm	12	Married-civ-spouse	Protective-serv	Husband	White	Male	0	0	40	United-States	>50K	Government	Married	Associate
3	44	Private	160323	Some-college	10	Married-civ-spouse	Machine-op-inspct	Husband	Black	Male	7688	0	40	United-States	>50K	Private	Married	Secodary
5	34	Private	198693	10th	6	Never-married	Other-service	Not-in-family	White	Male	0	0	30	United-States	<=50K	Private	Single	Secodary
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
48837	27	Private	257302	Assoc-acdm	12	Married-civ-spouse	Tech-support	Wife	White	Female	0	0	38	United-States	<=50K	Private	Married	Associate
48838	40	Private	154374	HS-grad	9	Married-civ-spouse	Machine-op-inspct	Husband	White	Male	0	0	40	United-States	>50K	Private	Married	Secodary
48839	58	Private	151910	HS-grad	9	Widowed	Adm-clerical	Unmarried	White	Female	0	0	40	United-States	<=50K	Private	Widowed	Secodary
48840	22	Private	201490	HS-grad	9	Never-married	Adm-clerical	Own-child	White	Male	0	0	20	United-States	<=50K	Private	Single	Secodary
48841	52	Self-emp-inc	287927	HS-grad	9	Married-civ-spouse	Exec-managerial	Wife	White	Female	15024	0	40	United-States	>50K	Self-Employed	Married	Secodary

45222 rows × 18 columns

Figure 5.2 Removing Missing Data from DataFrame

After removing the missing data from census, the DataFrame contains 45,222 rows. That is, we had a total of 3,620 observations that contain missing values originally.

1.2.2 Reducing Number of Categories

If a DataFrame contains categorical variables, they must be treated differently in comparison to scale or interval variables. In data analytics, we usually convert them to dummy variables in the pre-processing stage. We will discuss the conversion process in detail in Chapter 1.2.5. Nevertheless, if the variable contains a large number of categories, the number of dummy variables will become large as well. As a result, the analytics algorithm will have to handle a large number of variables and the required computational effort in the fitting process could be significant. One possible solution here is to reduce the number of categories at the expense of information loss. It is therefore a task for data analysts to balance this trade-off carefully.

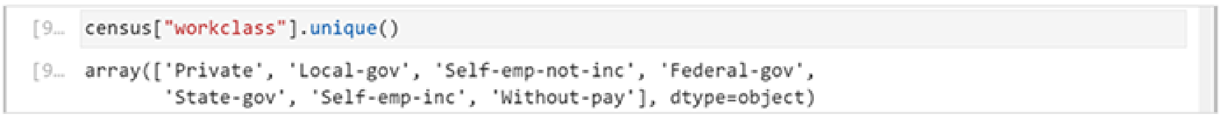
Basically, the process of category reduction is to put observations from similar categories into a new category. For instance, if the country names are categories of a categorical variable, we can group them by their continents, and if a categorical variable contains the models of a certain product, we can group them by their brands or their main features. The similarity of the categories is essential here for not losing too much information.

In terms of programming, all we need to do is to replace some category labels in a variable by the `.replace()` method of the pandas package.

```
DataFrame_Name["column_label"].replace(to_replace, value)
```

The parameter `to_replace` can be a list or dictionary of category labels to be replaced by the list or dictionary of new labels that is assigned to the `value` parameter.

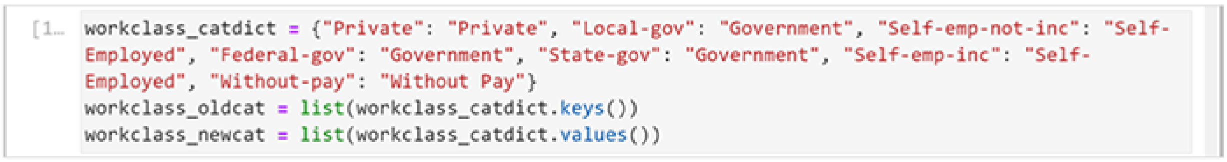
Example (Cont'd): In the US Adult Census dataset, there are three relevant categorical variables that we would like to reduce their number of categories: `workclass`, `occupation`, and `education` (`native-country` may have the most categories among all variables, but it is rather irrelevant). In the first step, we list out the categories of `workclass`.



```
[9… census["workclass"].unique()
[9… array(['Private', 'Local-gov', 'Self-emp-not-inc', 'Federal-gov',
           'State-gov', 'Self-emp-inc', 'Without-pay'], dtype=object)
```

Figure 5.3 Listing Out Unique Categories of a Variable

For instance, we can group all the government employees into one group and those in self-employment into another. To assign new labels to the old ones, we first create a dictionary in which the keys contain the original labels, and the values represent the new ones.

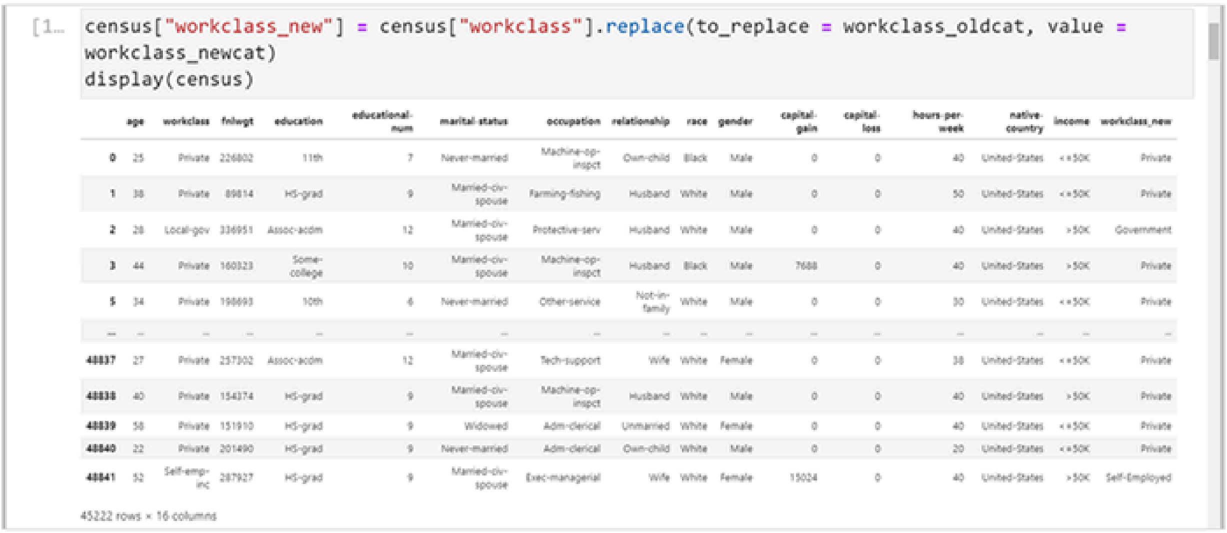


```
workclass_catdict = {"Private": "Private", "Local-gov": "Government", "Self-emp-not-inc": "Self-
Employed", "Federal-gov": "Government", "State-gov": "Government", "Self-emp-inc": "Self-
Employed", "Without-pay": "Without Pay"}
workclass_oldcat = list(workclass_catdict.keys())
workclass_newcat = list(workclass_catdict.values())
```

Figure 5.4 Creating a Dictionary with Old and New Category Labels

Actually, it is not required to construct this dictionary first. We could have also put the corresponding labels in the `.replace()` method directly. Although this step may seem redundant in the first sight, the advantage here is that we can refer to this dictionary to retrieve the original labels whenever the program requires. By applying the `.keys()` and `values()` methods to the dictionary, we can convert all the keys and values into their own list for the use in the `.replace()` method.

To maintain the possibility of using the original variable in the algorithms eventually, we save the variable with reduced categories as a new column and label it `workclass_new`.



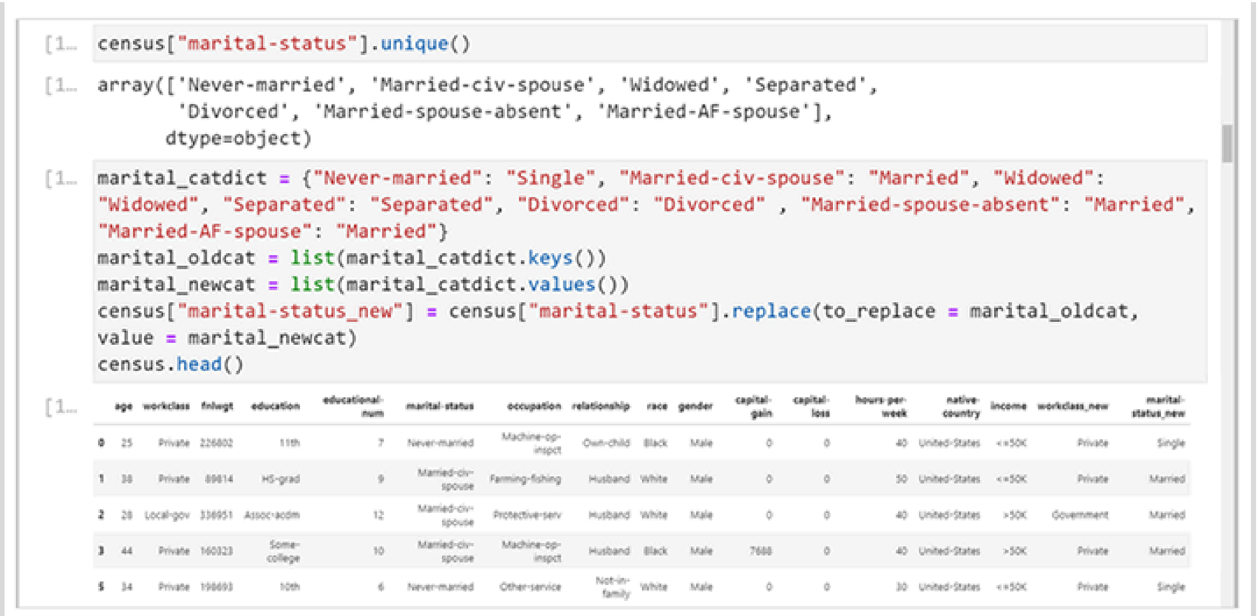
```
census["workclass_new"] = census["workclass"].replace(to_replace = workclass_oldcat, value =
workclass_newcat)
display(census)
```

	age	workclass	fnlwgt	education	educational-num	marital-status	occupation	relationship	race	gender	capital-gain	capital-loss	hours-per-week	native-country	income	workclass_new
0	25	Private	226802	11th	7	Never-married	Machine-op-inspct	Own-child	Black	Male	0	0	40	United-States	<=50K	Private
1	38	Private	89814	HS-grad	9	Married-civ-spouse	Farming-fishing	Husband	White	Male	0	0	50	United-States	<=50K	Private
2	28	Local-gov	336951	Assoc-acdm	12	Married-civ-spouse	Protective-serv	Husband	White	Male	0	0	40	United-States	>50K	Government
3	44	Private	160323	Some-college	10	Married-civ-spouse	Machine-op-inspct	Husband	Black	Male	7688	0	40	United-States	>50K	Private
5	34	Private	198693	10th	6	Never-married	Other-service	Not-in-family	White	Male	0	0	30	United-States	<=50K	Private
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
48837	27	Private	257302	Assoc-acdm	12	Married-civ-spouse	Tech-support	Wife	White	Female	0	0	38	United-States	<=50K	Private
48838	40	Private	154374	HS-grad	9	Married-civ-spouse	Machine-op-inspct	Husband	White	Male	0	0	40	United-States	>50K	Private
48839	58	Private	151910	HS-grad	9	Widowed	Adm-clerical	Unmarried	White	Female	0	0	40	United-States	<=50K	Private
48840	22	Private	201490	HS-grad	9	Never-married	Adm-clerical	Own-child	White	Male	0	0	20	United-States	<=50K	Private
48841	52	Self-emp-inc	287927	HS-grad	9	Married-civ-spouse	Exec-managerial	Wife	White	Female	15024	0	40	United-States	>50K	Self-Employed

45222 rows × 16 columns

Figure 5.5 Reducing the Number of Categories for a Categorical Variable

The newly created variable will appear at the rightmost column of the DataFrame. For `marital-status`, we follow the same steps to create a new column named `marital-status_new` with less categories.



```
[1… census["marital-status"].unique()

[1… array(['Never-married', 'Married-civ-spouse', 'Widowed', 'Separated',
           'Divorced', 'Married-spouse-absent', 'Married-AF-spouse'],
          dtype=object)

[1… marital_catdict = {"Never-married": "Single", "Married-civ-spouse": "Married", "Widowed":
    "Widowed", "Separated": "Separated", "Divorced": "Divorced" , "Married-spouse-absent": "Married",
    "Married-AF-spouse": "Married"}
    marital_oldcat = list(marital_catdict.keys())
    marital_newcat = list(marital_catdict.values())
    census["marital-status_new"] = census["marital-status"].replace(to_replace = marital_oldcat,
    value = marital_newcat)
    census.head()
```

[1…

	age	workclass	fnlwgt	education	educational-num	marital-status	occupation	relationship	race	gender	capital-gain	capital-loss	hours-per-week	native-country	income	workclass_new	marital-status_new
0	25	Private	226802	11th	7	Never-married	Machine-op-inspct	Own-child	Black	Male	0	0	40	United-States	<=50K	Private	Single
1	38	Private	89814	HS-grad	9	Married-civ-spouse	Farming-fishing	Husband	White	Male	0	0	50	United-States	<=50K	Private	Married
2	28	Local-gov	336951	Assoc-acdm	12	Married-civ-spouse	Protective-serv	Husband	White	Male	0	0	40	United-States	>50K	Government	Married
3	44	Private	160323	Some-college	10	Married-civ-spouse	Machine-op-inspct	Husband	Black	Male	7688	0	40	United-States	>50K	Private	Married
5	34	Private	198693	10th	6	Never-married	Other-service	Not-in-family	White	Male	0	0	30	United-States	<=50K	Private	Single

Figure 5.6 Reducing the Number of Categories for a Categorical Variable

In Figure 5.6, we can see that the different marriage statuses have been merged to a general group named "`Married`". Furthermore, the label of the category "`Never-married`" has been shortened to "`Single`" to favour the visualisation of the result output later.

To reduce the categories in `education`, we can transform `educational-num` instead since they are equivalent. Moreover, `educational-num` is an ordered numeric variable so that we can apply discretisation as described in the following section.

Read

Refer to the link below for more details and examples on the `.replace()` method of the pandas package:

https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.replace.html

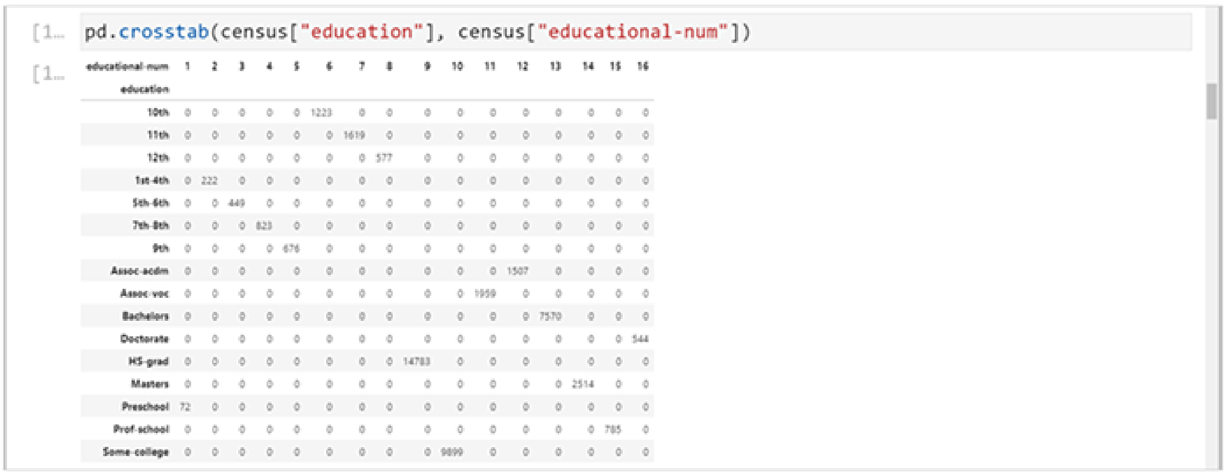
1.2.3 Discretisation

If a categorical variable has ordered numeric values as categories, we can discretise them into new bins by the `cut()` function of the pandas package. The `cut()` function has been introduced in Chapter 5.2 of Study Unit 4.

```
DataFrame_name["column"] = pd.cut(x = array, bins, labels)
```

For our purpose here to reduce the number of categories, it is sufficient to put the highest value of each category in the list assigned to `bins`. When applying the `cut()` function, we have to be aware that it only includes the rightmost edge in each bin and not the leftmost one. Hence, the list for bins should start with 0, or -1 in case 0 is one of the numeric values of the original categories.

Example (Cont'd): As mentioned in the previous section, the variables `education` and `educational-num` are equivalent. We can show this by the `crosstab()` function where the frequency of each category in `education` will be counted when cross-combining with a value in `educational-num`.



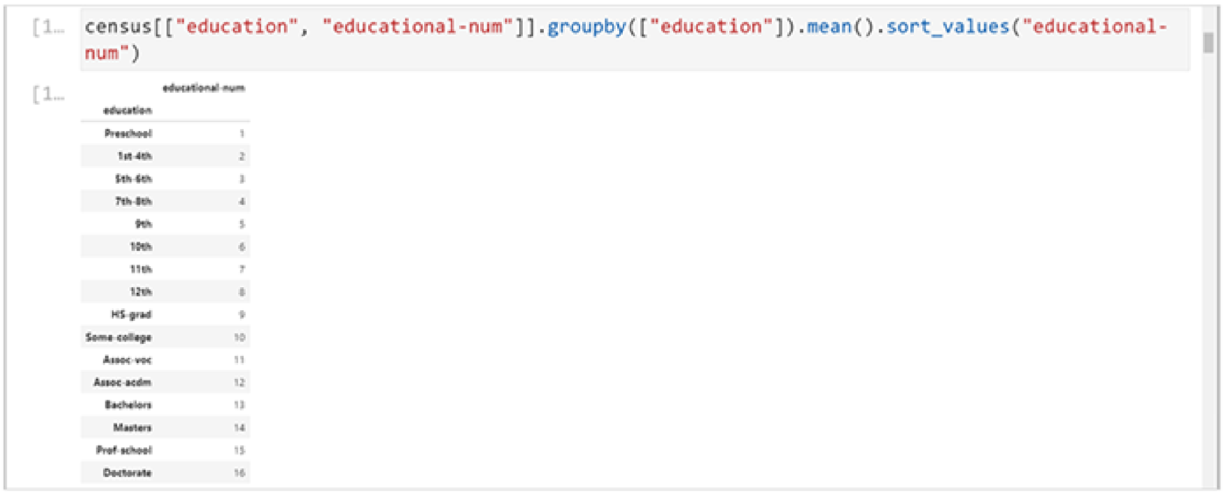
```
[1… pd.crosstab(census["education"], census["educational-num"])
```

[1…

educational-num	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
education																
10th	0	0	0	0	0	1223	0	0	0	0	0	0	0	0	0	0
11th	0	0	0	0	0	0	1619	0	0	0	0	0	0	0	0	0
12th	0	0	0	0	0	0	0	577	0	0	0	0	0	0	0	0
1st-4th	0	222	0	0	0	0	0	0	0	0	0	0	0	0	0	0
5th-6th	0	0	449	0	0	0	0	0	0	0	0	0	0	0	0	0
7th-8th	0	0	0	823	0	0	0	0	0	0	0	0	0	0	0	0
9th	0	0	0	0	676	0	0	0	0	0	0	0	0	0	0	0
Assoc-acdm	0	0	0	0	0	0	0	0	0	0	0	1507	0	0	0	0
Assoc-voc	0	0	0	0	0	0	0	0	0	0	1959	0	0	0	0	0
Bachelors	0	0	0	0	0	0	0	0	0	0	0	0	7570	0	0	0
Doctorate	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	544
HS-grad	0	0	0	0	0	0	0	0	14783	0	0	0	0	0	0	0
Masters	0	0	0	0	0	0	0	0	0	0	0	0	0	2514	0	0
Preschool	72	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Prof-school	0	0	0	0	0	0	0	0	0	0	0	0	0	0	785	0
Some-college	0	0	0	0	0	0	0	0	0	9899	0	0	0	0	0	0

Figure 5.7 Cross Tabulation of Two Categorical Variables

From Figure 5.7, we can see that there are only counts in one category in `education` for each value in `educational-num`. In other words, every category in `educational-num` corresponds to exactly one category in `education`. Furthermore, we can also see that the education level increases with the increase of the value in `educational-num`. For instance, "`pre-school`" level is 1 in `educational-num` while education level "`1st-4th`" has value 2. The highest education level in this dataset is "`Doctorate`" and its value in `educational-num` is also the highest, namely 16. To gain a better overview of the categories in both variables, we can chain the methods `.groupby().mean().sort_values()` as in Figure 5.8.



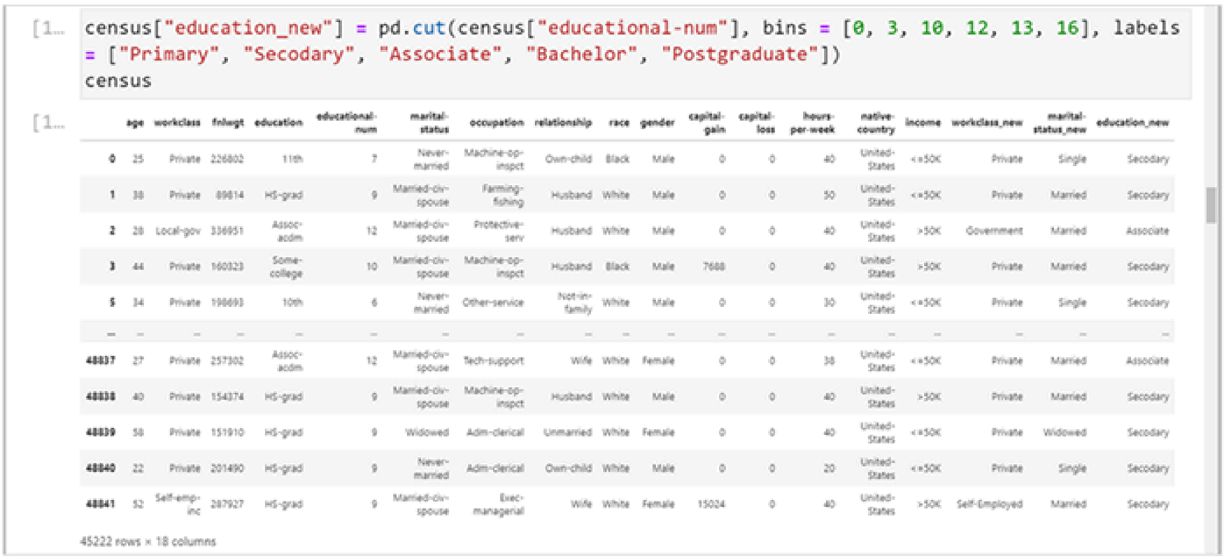
```
[1… census[["education", "educational-num"]].groupby(["education"]).mean().sort_values("educational-
    num")
```

```
[1…              educational-num
      education
      Preschool                1
        1st-4th                2
        5th-6th                3
        7th-8th                4
            9th                5
           10th                6
           11th                7
           12th                8
        HS-grad                9
   Some-college               10
      Assoc-voc               11
     Assoc-acdm               12
      Bachelors               13
        Masters               14
    Prof-school               15
      Doctorate               16
```

Figure 5.8 Numeric Labels of a Categorical Variable

In the first step, we select the only required columns `education` and `educational-num` for this output. Then we use the `.groupby()` method to group the variables by the categories in `education`. At the same time, the mean for each category in `educational-num` will be calculated. Since the values in `educational-num` are the same for all observations in the same `education` category, their mean must be equal to their category value in `educational-num`. In the final step, we sort the rows of the grouped table by the values in `educational-num` since it is the ordered version of all the educational levels.

Eventually, we can reduce the number of categories in `education` by assigning the values in `educational-num` to new bins.



```
[1… census["education_new"] = pd.cut(census["educational-num"], bins = [0, 3, 10, 12, 13, 16], labels
    = ["Primary", "Secodary", "Associate", "Bachelor", "Postgraduate"])
    census
```

[1…

	age	workclass	fnlwgt	education	educational-num	marital-status	occupation	relationship	race	gender	capital-gain	capital-loss	hours-per-week	native-country	income	workclass_new	marital-status_new	education_new
0	25	Private	226802	11th	7	Never-married	Machine-op-inspct	Own-child	Black	Male	0	0	40	United-States	<=50K	Private	Single	Secodary
1	38	Private	89814	HS-grad	9	Married-civ-spouse	Farming-fishing	Husband	White	Male	0	0	50	United-States	<=50K	Private	Married	Secodary
2	28	Local-gov	336951	Assoc-acdm	12	Married-civ-spouse	Protective-serv	Husband	White	Male	0	0	40	United-States	>50K	Government	Married	Associate
3	44	Private	160323	Some-college	10	Married-civ-spouse	Machine-op-inspct	Husband	Black	Male	7688	0	40	United-States	>50K	Private	Married	Secodary
5	34	Private	198693	10th	6	Never-married	Other-service	Not-in-family	White	Male	0	0	30	United-States	<=50K	Private	Single	Secodary
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
48837	27	Private	257302	Assoc-acdm	12	Married-civ-spouse	Tech-support	Wife	White	Female	0	0	38	United-States	<=50K	Private	Married	Associate
48838	40	Private	154374	HS-grad	9	Married-civ-spouse	Machine-op-inspct	Husband	White	Male	0	0	40	United-States	>50K	Private	Married	Secodary
48839	58	Private	151910	HS-grad	9	Widowed	Adm-clerical	Unmarried	White	Female	0	0	40	United-States	<=50K	Private	Widowed	Secodary
48840	22	Private	201490	HS-grad	9	Never-married	Adm-clerical	Own-child	White	Male	0	0	20	United-States	<=50K	Private	Single	Secodary
48841	52	Self-emp-inc	287927	HS-grad	9	Married-civ-spouse	Exec-managerial	Wife	White	Female	15024	0	40	United-States	>50K	Self-Employed	Married	Secodary

45222 rows × 18 columns

Figure 5.9 Discretising an Ordered Numeric Categorical Variable

In `education_new`, we summarise "`pre-school`" (1) and all the primary (2-3) levels to the category "`Primary`". All the secondary levels, including those who could not complete the college degree (4-10) are assigned to "`Secondary`". Both "`Assoc-voc`" (11) and "`Assoc-acdm`" (12) are now in the group of "`Associate`". While "`Bachelor`" (13) remains "`Bachelor`", every observation with higher education levels than that (14-16) is now grouped into the category "`Postgraduate`".

Since 0 does not belong to the value in `educational-num`, we can include it in the list assigned to the parameter `bins` and take it as the leftmost edge of the first bin.

Read

Refer to the link below for more details and examples on the `crosstab()` function of the pandas package:

https://pandas.pydata.org/docs/reference/api/pandas.crosstab.html

1.2.4 Selecting and Renaming Variables

It is very common that a dataset contains variables that are not directly relevant to be included in the analytics algorithm. Some of them could be redundant in their meaning; some of them are the original version of a transformed variable. These variables should be removed from the DataFrame before using the data to run the scikit-learn estimator.

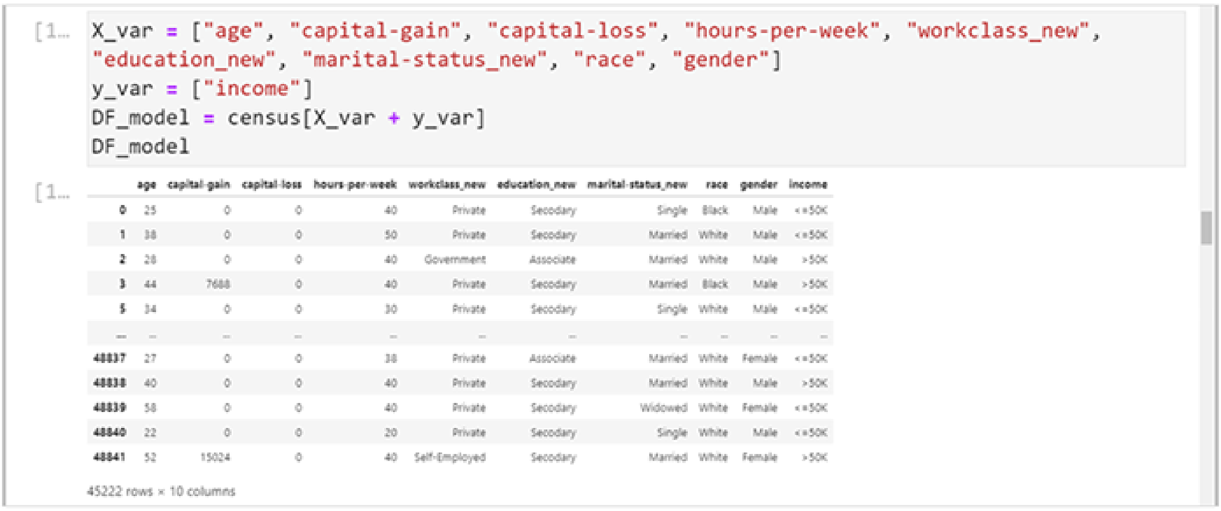
In Chapter 2 of Study Unit 4, we have learned how to select rows and columns from the pandas DataFrame using index, Boolean masks, and localisation. The attributes `.iloc()` and `.loc()` are crucial in this context. We can use the same procedures to select the necessary columns for the scikit-learn algorithm.

If a dataset is originated from an external source, the given variable names may not necessarily reflect the needs and ideas of the analyst. Sometimes, they can be lengthy and make the result output visually appalling. In pandas, the `.rename()` method is used to rename the variables in a DataFrame.

```
DataFrame_name.rename(columns = {"oldvar": "newvar"})
```

The column labels to be renamed must be put as keys of a dictionary that will be assigned to the parameter `columns` in the `.rename()` method. The values of the dictionary will then be the new labels of the corresponding columns.

Example (Cont'd): After reducing the number of categories in `workclass`, `marital-status` and `education`, we have variables of both versions in our DataFrame. For the construction of the model, however, we only need the new ones. That is, we shall only select all categorical variables with the suffix "`_new`". Furthermore, `fnlwgt` is not relevant for our analyses and will be dropped. The variables `occupation`, `relationship` and `native-country` are not included in the models as well since they are correlated with the variables `workclass`, `marital-status` and `race`, respectively. After discretisation, `educational-num` also becomes redundant. Hence, it will be removed from the final selection.



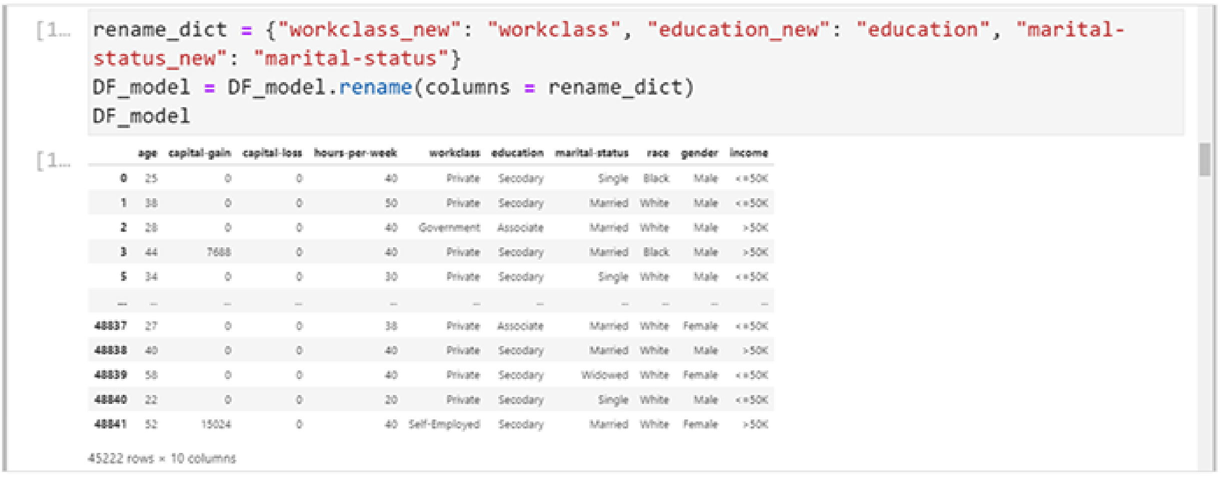
```
[1… X_var = ["age", "capital-gain", "capital-loss", "hours-per-week", "workclass_new",
    "education_new", "marital-status_new", "race", "gender"]
    y_var = ["income"]
    DF_model = census[X_var + y_var]
    DF_model
```

```
[1…         age  capital-gain  capital-loss  hours-per-week  workclass_new  education_new  marital-status_new   race  gender  income
        0    25             0             0              40        Private       Secodary              Single  Black    Male   <=50K
        1    38             0             0              50        Private       Secodary             Married  White    Male   <=50K
        2    28             0             0              40     Government      Associate             Married  White    Male    >50K
        3    44          7688             0              40        Private       Secodary             Married  Black    Male    >50K
        5    34             0             0              30        Private       Secodary              Single  White    Male   <=50K
      ...   ...           ...           ...             ...            ...            ...                 ...    ...     ...     ...
    48837    27             0             0              38        Private      Associate             Married  White  Female   <=50K
    48838    40             0             0              40        Private       Secodary             Married  White    Male    >50K
    48839    58             0             0              40        Private       Secodary             Widowed  White  Female   <=50K
    48840    22             0             0              20        Private       Secodary              Single  White    Male   <=50K
    48841    52         15024             0              40  Self-Employed       Secodary             Married  White  Female    >50K

    45222 rows × 10 columns
```

Figure 5.10 Selecting Relevant Variables

We first create a list of the independent variables named `X_var` and a list with the dependent variable named `y_var`. They will then be concatenated before being selected from the `census` DataFrame by indexing. The resulting DataFrame is named `DF_model`.

Since we have removed `workclass`, `marital-status`, and `education` from the `DF_model` DataFrame, we can now rename `workclass_new`, `marital-status_new`, and `education_new` back to the names of the original variables.



```
[1… rename_dict = {"workclass_new": "workclass", "education_new": "education", "marital-
    status_new": "marital-status"}
    DF_model = DF_model.rename(columns = rename_dict)
    DF_model
```

```
[1…         age  capital-gain  capital-loss  hours-per-week      workclass  education  marital-status   race  gender  income
        0    25             0             0              40        Private   Secodary          Single  Black    Male   <=50K
        1    38             0             0              50        Private   Secodary         Married  White    Male   <=50K
        2    28             0             0              40     Government  Associate         Married  White    Male    >50K
        3    44          7688             0              40        Private   Secodary         Married  Black    Male    >50K
        5    34             0             0              30        Private   Secodary          Single  White    Male   <=50K
      ...   ...           ...           ...             ...            ...        ...             ...    ...     ...     ...
    48837    27             0             0              38        Private  Associate         Married  White  Female   <=50K
    48838    40             0             0              40        Private   Secodary         Married  White    Male    >50K
    48839    58             0             0              40        Private   Secodary         Widowed  White  Female   <=50K
    48840    22             0             0              20        Private   Secodary          Single  White    Male   <=50K
    48841    52         15024             0              40  Self-Employed   Secodary         Married  White  Female    >50K

    45222 rows × 10 columns
```

Figure 5.11 Renaming Selected Variables

Read

Refer to the link below for more details and examples on the `.rename()` method of the pandas package:

https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.rename.html

1.2.5 Creating Dummy Variables

As mentioned, categorical variables must be converted to dummy variables before they can be evaluated and included in the computation of the scikit-learn algorithms. Dummy variables are binary variables that only have two values: 0 and 1. If an observation belongs to a certain category, the corresponding dummy variable will be 1, otherwise 0. Since each category of a categorical variable will be transformed to a dummy variable, the number of categories has indeed a direct impact on the number of dummy variables in the final DataFrame used in the algorithm. As a result, it is important to keep the number of categories at a rather low level. The syntaxes and procedures for category reduction have been discussed in Chapter 1.2.2 and 1.2.3 of this study unit.

In Python, we can convert categorical variables to dummy variables using the `get_dummies()` function of the pandas package.

```
DataFrame_name["column"] = pd.get_dummies(data, drop_first)
```

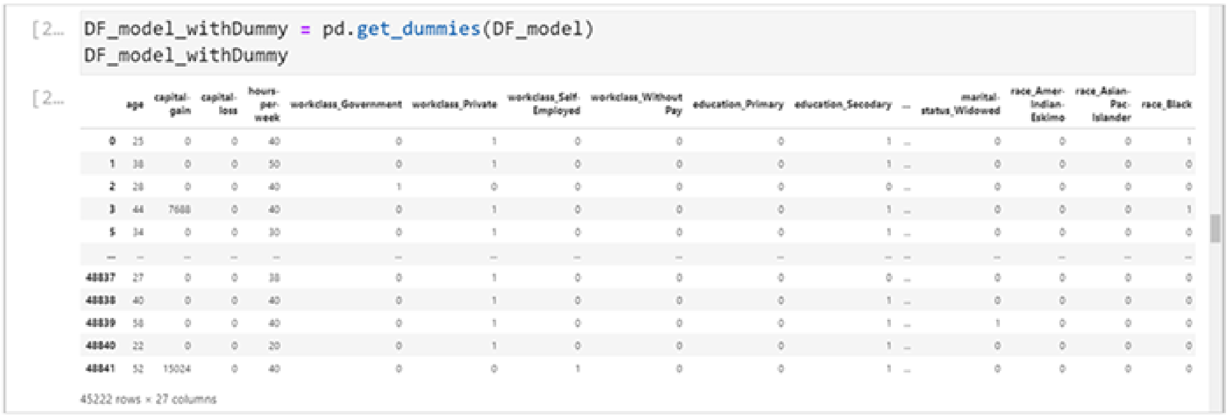
The parameter `drop_first` is used to instruct Python to take the first category as the reference level and remove it from the resulting DataFrame. The reason to define a reference level for categorical variables and to remove it from the DataFrame is to avoid linear dependence in the data matrix, which causes error in the calculation. The default setting here is `drop_first = False`. In this case, all dummy variables will remain in the resulting DataFrame. Since most of the modules in scikit-learn have their own algorithms to deal with this issue, we can keep this setting without causing error in the estimation process.

The meaning of the parameter `data` is obvious. Note that pandas will create dummy variables for each uniquely existing string in all non-numeric variables. If we have a numeric categorical variable which we would like to convert to dummy variables as well, we will need to change its data type using the `.astype()` method.

```
DataFrame_name.astype({"var_name": "type_str", …})
```

A dictionary should be assigned to the .astype() method with the variable names as the keys and the data types as the values, which can be `"int"`, `"float"`, `"category"`, `"str"`, `"bool"`, etc. To create dummy variables for a numeric categorical variable, we can either change the type of the orginal variable to `"category"` or `"str"`.

Example (Cont'd): For the categorical variables `workclass`, `marital-status`, `education`, `race` and `gender`, dummy variables must be created before they can be included in the construction of the analytics models.



```
[2… DF_model_withDummy = pd.get_dummies(DF_model)
    DF_model_withDummy
```

[2…

	age	capital-gain	capital-loss	hours-per-week	workclass_Government	workclass_Private	workclass_Self-Employed	workclass_Without Pay	education_Primary	education_Secodary	...	marital-status_Widowed	race_Amer-Indian-Eskimo	race_Asian-Pac-Islander	race_Black
0	25	0	0	40	0	1	0	0	0	1	...	0	0	0	1
1	38	0	0	50	0	1	0	0	0	1	...	0	0	0	0
2	28	0	0	40	1	0	0	0	0	0	...	0	0	0	0
3	44	7688	0	40	0	1	0	0	0	1	...	0	0	0	1
5	34	0	0	30	0	1	0	0	0	1	...	0	0	0	0
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
48837	27	0	0	38	0	1	0	0	0	0	...	0	0	0	0
48838	40	0	0	40	0	1	0	0	0	1	...	0	0	0	0
48839	58	0	0	40	0	1	0	0	0	1	...	1	0	0	0
48840	22	0	0	20	0	1	0	0	0	1	...	0	0	0	0
48841	52	15024	0	40	0	0	1	0	0	1	...	0	0	0	0

45222 rows × 27 columns

Figure 5.12 Creating Dummy Variables from Categorical Variables

In Figure 5.12, the numeric variables remain unchanged as they do not have the appropriate data type to be converted. The categorical variables, on the other hand, have now been replaced by the dummy variables entirely. Therefore, we recommend carrying out such transformation in a new DataFrame so that we do not lose the original data in case anything goes wrong.

Read

Refer to the link below for more details and examples on the `get_dummies()` function of the pandas package:

https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.get_dummies.html

Refer to the link below for more details and examples on the `.astype()` method of the pandas package:

https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.astype.html

1.2.6 Data Transformation

In the previous sections, we have discussed how to prepare categorical variables for scikit-learn analytics algorithms. Though numeric variables are generally easier to deal with, they may also cause trouble in the model estimation. For instance, their range of values can be rather wide. As a result, variables with such characteristic tend to have higher impact in the model than those with smaller value ranges. Hence, they need to be scaled down to match the value range of the other numeric variables. The most common methods for this purpose are normalisation and standardisation.

In Chapter 5.4 of Study Unit 4, we have discussed how to execute log-transformation, standardisation, and normalisation on pandas DataFrames. However, we have had to construct our own programs for standardisation and normalisation since they are not included in pandas. In scikit-learn, on the other hand, these functions can be found in the `preprocessing` module, where the `normalize()` function is actually straightforward and easy to handle.

```
Object_name = sklearn.preprocessing.normalize(X)
```

The object assigned to parameter `X` can be a DataFrame, a NumPy array, etc.

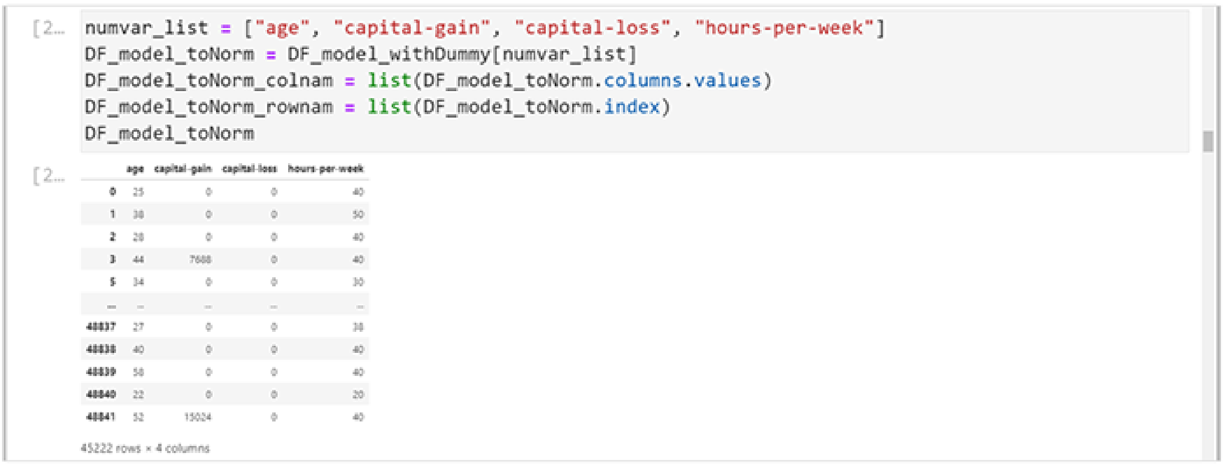
Conversely, the syntax for standardisation requires the initialisation of the estimator first, the data can then be transformed in the subsequent step.

```
scaler = preprocessing.StandardScaler()
scaler.fit(X)
Object_name = scaler.transform(X)
```

We first initiate `scaler` as the estimator object for `StandardScaler` from the `preprocessing` module. Then, the mean and standard deviation of the object `X`, which is usually an Array or a DataFrame, will be computed by the `fit()` function. In the final step, the object `X` will be standardised by the `transform()` function.

Example (Cont'd): In view of K-Means clustering that we will be carrying out in the next Chapter, we need to normalise the numeric variables. In particular, the values of the variables `capital-gain` and `capital-loss` have very wide range (0 - 99,999 and 0 - 4,356, respectively) and may affect the fit or the explanatory power of the model.

In the first step, we select those columns that should be normalised and save it in a new DataFrame. The reason of generating a subset DataFrame here is that `normalize()` will generate a function which transforms all numeric variables including the dummy variables created from the categorical data. Though their values will remain 0 and 1 after normalisation, we can shorten the processing time by not letting irrelevant variables involved in the process.

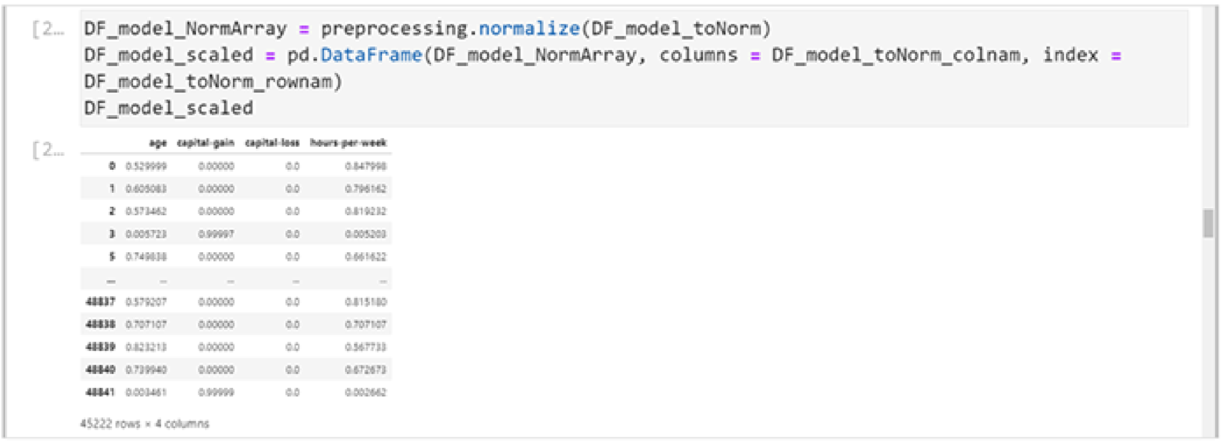


```
[2… numvar_list = ["age", "capital-gain", "capital-loss", "hours-per-week"]
    DF_model_toNorm = DF_model_withDummy[numvar_list]
    DF_model_toNorm_colnam = list(DF_model_toNorm.columns.values)
    DF_model_toNorm_rownam = list(DF_model_toNorm.index)
    DF_model_toNorm
```

```
[2…          age  capital-gain  capital-loss  hours-per-week
         0    25             0             0              40
         1    38             0             0              50
         2    28             0             0              40
         3    44          7688             0              40
         5    34             0             0              30
       ...   ...           ...           ...             ...
     48837    27             0             0              38
     48838    40             0             0              40
     48839    58             0             0              40
     48840    22             0             0              20
     48841    52         15024             0              40

    45222 rows × 4 columns
```

Figure 5.13 Selecting Numeric Variables for Normalisation

In the first line, we create a list of variables to be normalised and subset the original DataFrame using this list. Since the resulting object of the `normalize()` function is a NumPy array, we will have to convert it back to a pandas DataFrame with column and row labels (Note: NumPy arrays have no labels on both axes). Therefore, we need to save the labels as two Python lists named `DF_model_toNorm_colnam` and `DF_model_toNorm_rownam` for later use.



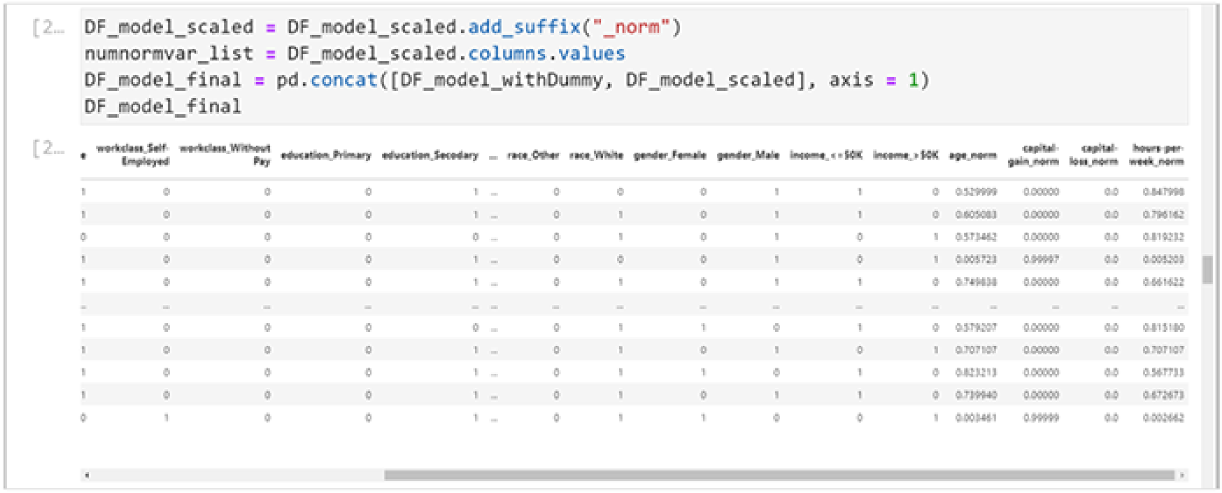
```
[2… DF_model_NormArray = preprocessing.normalize(DF_model_toNorm)
    DF_model_scaled = pd.DataFrame(DF_model_NormArray, columns = DF_model_toNorm_colnam, index =
    DF_model_toNorm_rownam)
    DF_model_scaled
```

```
[2…            age  capital-gain  capital-loss  hours-per-week
        0  0.529999      0.00000           0.0        0.847998
        1  0.605083      0.00000           0.0        0.796162
        2  0.573462      0.00000           0.0        0.819232
        3  0.005723      0.99997           0.0        0.005203
        5  0.749838      0.00000           0.0        0.661622
      ...       ...          ...           ...             ...
    48837  0.579207      0.00000           0.0        0.815180
    48838  0.707107      0.00000           0.0        0.707107
    48839  0.823213      0.00000           0.0        0.567733
    48840  0.739940      0.00000           0.0        0.672673
    48841  0.003461      0.99999           0.0        0.002662

    45222 rows × 4 columns
```

Figure 5.14 Normalising Numeric Variables

After the normalisation process, we can convert the array `DF_model_NormArray` resulting from the `preprocessing.normalize()` function to a pandas DataFrame by the `pd.DataFrame()` function. We can also set our own column and row labels by assigning the lists `DF_model_toNorm_colnam` and `DF_model_toNorm_rownam` to the `columns` and `index` parameters, respectively.

In the final step, we rename the normalised variables and append them to the original DataFrame.



```
[2… DF_model_scaled = DF_model_scaled.add_suffix("_norm")
    numnormvar_list = DF_model_scaled.columns.values
    DF_model_final = pd.concat([DF_model_withDummy, DF_model_scaled], axis = 1)
    DF_model_final
```

```
[2… e  workclass_Self-  workclass_Without  education_Primary  education_Secodary  ...  race_Other  race_White  gender_Female  gender_Male  income_<=50K  income_>50K  age_norm  capital-   capital-   hours-per-
              Employed                Pay                                                                                                                                  gain_norm  loss_norm  week_norm
    1                0                  0                  0                   1  ...           0           0              0            1             1            0  0.529999    0.00000        0.0   0.847998
    1                0                  0                  0                   1  ...           0           1              0            1             1            0  0.605083    0.00000        0.0   0.796162
    0                0                  0                  0                   0  ...           0           1              0            1             0            1  0.573462    0.00000        0.0   0.819232
    1                0                  0                  0                   1  ...           0           0              0            1             0            1  0.005723    0.99997        0.0   0.005203
    1                0                  0                  0                   1  ...           0           1              0            1             1            0  0.749838    0.00000        0.0   0.661622
    ...            ...                ...                ...                 ...  ...         ...         ...            ...          ...           ...          ...       ...        ...        ...        ...
    1                0                  0                  0                   0  ...           0           1              1            0             1            0  0.579207    0.00000        0.0   0.815180
    1                0                  0                  0                   1  ...           0           1              0            1             0            1  0.707107    0.00000        0.0   0.707107
    1                0                  0                  0                   1  ...           0           1              1            0             1            0  0.823213    0.00000        0.0   0.567733
    1                0                  0                  0                   1  ...           0           1              0            1             1            0  0.739940    0.00000        0.0   0.672673
    0                1                  0                  0                   1  ...           0           1              1            0             0            1  0.003461    0.99999        0.0   0.002662
```

Figure 5.15 Concatenating Normalised Variables with the Original DataFrame

Note that we have used the original variable names as the column labels of the normalised variables. If we simply concatenate these DataFrames, these labels would be duplicate. Hence, we need to append "`_norm`" as suffix to all labels of the normalised variables by the `.add_suffix()` method. Only after ensuring that all labels in the concatenated DataFrames are unique, we can merge the DataFrames. Furthermore, we have saved the names of the normalised variables with the suffix "`_norm`" in a new list named `numnormvar_list` just in case we will need to extract them again from the DataFrame in the future.

Read

Refer to the link below for more details and examples on the `normalize()` function of the `preprocessing` module in the scikit-learn package:

https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.normalize.html

Refer to the link below for more details and examples on the `sklearn.#preprocessing.StandardScaler` algorithm of the scikit-learn package:

https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.StandardScaler.html

Refer to the link below for more details and examples on the `.add_suffix()` method of the pandas package:

https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.add_suffix.html

1.2.7 Splitting DataFrames for Training and Testing

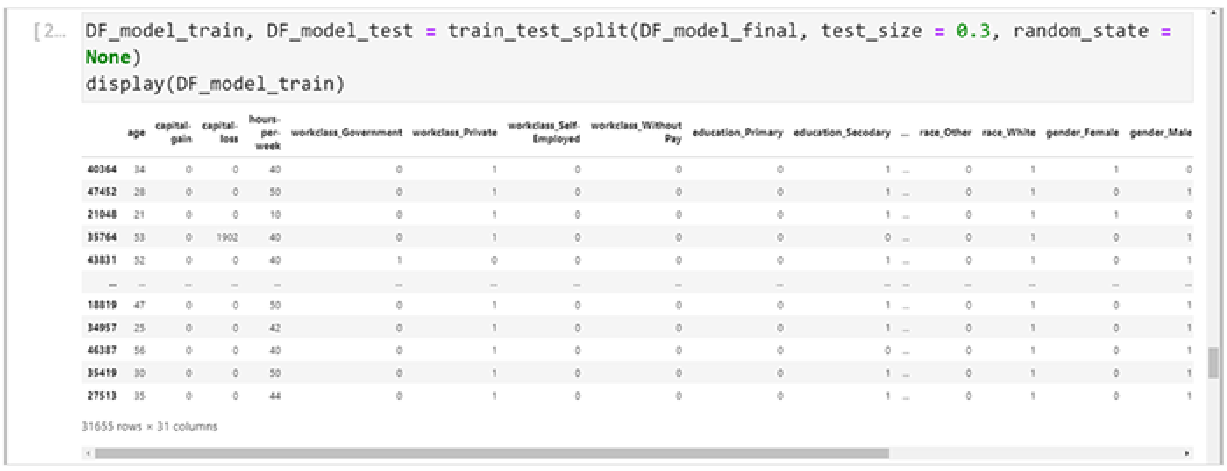
In data analytics, the performance of a predictive model can be measured by its accuracy of predicting unseen data. However, such data are usually unavailable. On the other hand, testing the prediction performance of a model by applying it on the original data based on which the model was constructed in the first place is not sensible at all. Therefore, analysts usually "hold back" a subset of data, namely the testing dataset, for model evaluation purpose. The remaining data form the training dataset for the construction of the model.

In Python, the module `model_selection` provides the `train_test_split()` function to draw observations randomly from the original DataFrame into the training and the testing datasets.

```
Object_name = sklearn.model_selection.train_test_split(arrays,
                        test_size, random_state)
```

The objects assigned to the parameter `arrays` can be NumPy arrays, pandas DataFrames, etc. A value between 0 and 1 should be given to the parameter `test_size`, which determines the proportion of observations in the original array that should be distributed to the testing dataset. The default value here is 0.25. The parameter `random_state` controls the shuffling of the data before applying the split. The default value here is "`None`", which means that different random seeds will be selected every time the function is being executed. That is, different observations will be chosen for the testing dataset in every run. Consequently, the testing result of the model will be different as well.

Example (Cont'd): After all the data preparation steps have been carried out, we can now split the DataFrame into a training and a testing dataset. The testing data proportion we require here is 30%, and we will set `random_state` to "`None`".



```
DF_model_train, DF_model_test = train_test_split(DF_model_final, test_size = 0.3, random_state =
None)
display(DF_model_train)
```

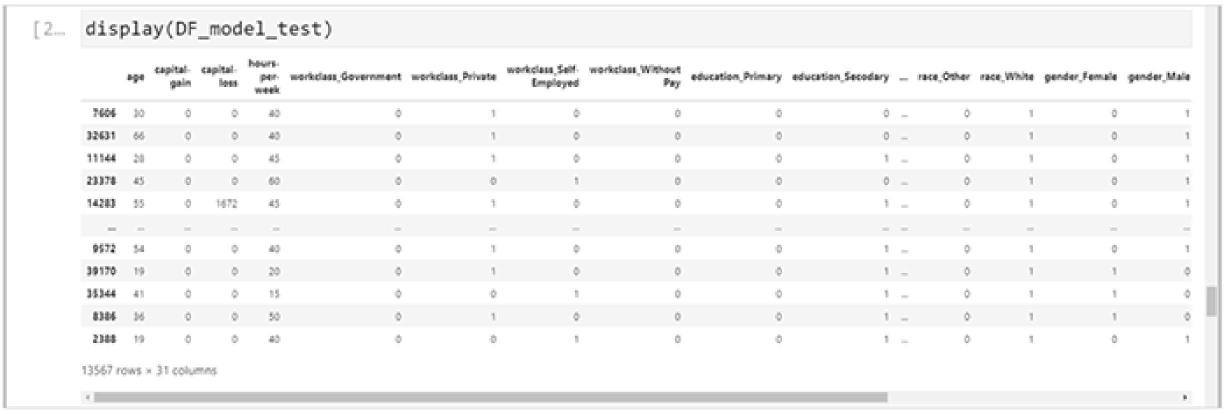
	age	capital-gain	capital-loss	hours-per-week	workclass_Government	workclass_Private	workclass_Self-Employed	workclass_Without Pay	education_Primary	education_Secodary	...	race_Other	race_White	gender_Female	gender_Male
40364	34	0	0	40	0	1	0	0	0	1	...	0	1	1	0
47452	28	0	0	50	0	1	0	0	0	1	...	0	1	0	1
21048	21	0	0	10	0	1	0	0	0	1	...	0	1	1	0
35764	53	0	1902	40	0	1	0	0	0	0	...	0	1	0	1
43831	52	0	0	40	1	0	0	0	0	1	...	0	1	0	1
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
18819	47	0	0	50	0	1	0	0	0	1	...	0	1	0	1
34957	25	0	0	42	0	1	0	0	0	1	...	0	1	0	1
46387	56	0	0	40	0	1	0	0	0	0	...	0	1	0	1
35419	30	0	0	50	0	1	0	0	0	1	...	0	1	0	1
27513	35	0	0	44	0	1	0	0	0	1	...	0	1	0	1

31655 rows × 31 columns

Figure 5.16 Training Dataset after Split from the Original DataFrame

We can see from Figure 5.16 that the training dataset contains 31,655 rows which are exactly 70% of the original 45,222 observations. Furthermore, the rows have been shuffled completely as indicated by the chaotic order of the row indices.

The remaining 13,567 rows are now assigned to the testing dataset as Figure 5.17 illustrates.



```
display(DF_model_test)
```

	age	capital-gain	capital-loss	hours-per-week	workclass_Government	workclass_Private	workclass_Self-Employed	workclass_Without Pay	education_Primary	education_Secodary	...	race_Other	race_White	gender_Female	gender_Male
7606	30	0	0	40	0	1	0	0	0	0	...	0	1	0	1
32631	66	0	0	40	0	1	0	0	0	0	...	0	1	0	1
11144	28	0	0	45	0	1	0	0	0	1	...	0	1	0	1
23378	45	0	0	60	0	0	1	0	0	0	...	0	1	0	1
14283	55	0	1672	45	0	1	0	0	0	1	...	0	1	0	1
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
9572	54	0	0	40	0	1	0	0	0	1	...	0	1	0	1
39170	19	0	0	20	0	1	0	0	0	1	...	0	1	1	0
35344	41	0	0	15	0	0	1	0	0	1	...	0	1	1	0
8386	36	0	0	50	0	1	0	0	0	1	...	0	1	1	0
2388	19	0	0	40	0	0	1	0	0	1	...	0	1	0	1

13567 rows × 31 columns

Figure 5.17 Testing Dataset after Split from the Original DataFrame

Read

Refer to the link below for more details and examples on the `train_test_split()` function of the `model_selection` module in the scikit-learn package:

https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.train_test_split.html

1.2.8 Extracting Dependent and Independent Variables

In scikit-learn, the ultimate command of many algorithms to fit a model on a DataFrame is the `.fit()` function. The `.fit()` function has usually two parameters: `X` and `Y`, where `Y` could be optional for some algorithms. The parameter `X` is the design matrix that contains all independent variables, and `Y` is the vector of the target variable. Both `X` and `Y` can be NumPy arrays or pandas DataFrames. As a result, we need to extract the independent variables as a matrix and the dependent variable as a vector from the original DataFrame.

The procedure here is rather straightforward. We simply select the column that represents our dependent variable and save it as a new object.

```
y = DataFrame_name["target_var"]
```

Similarly, the matrix of the independent variables can be selected in the same manner.

```
X = DataFrame_name[["X1", "X2", …]]
```

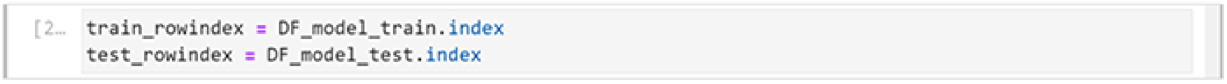
Note that it is required to wrap the names of the independent variables in a list (square brackets) first before putting them in the index operator `[]`.

If the DataFrame only contains the independent variables and the target variable, we can also simply drop the target variable from the original DataFrame to obtain X.

```
X = DataFrame_name.drop("target_var")
```

If the target variable is categorical and has been transformed to various dummy variables, the names of the dummy variables must be put in a list when passing them to the `.drop()` method.

Example (Cont'd): Before separating our training and testing datasets for the dependent and independent variables, we shall save the row indices of both the datasets in two new lists for later identification of the corresponding observations.

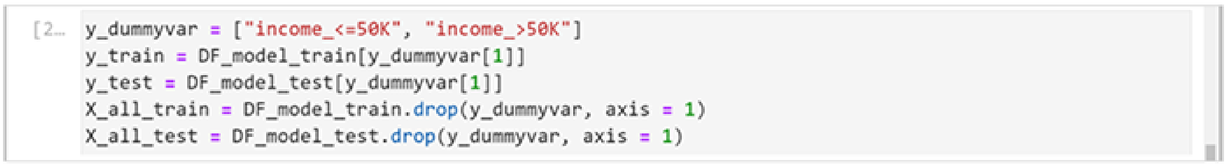


```
[2… train_rowindex = DF_model_train.index
    test_rowindex = DF_model_test.index
```

Figure 5.18 Saving the Row Indices in Python Lists

Since our DataFrames `DF_model_train` and `DF_model_test` only contain the independent and dependent variables, we can simply extract the dependent variables for `y_train` and `y_test` and drop them from the DataFrames subsequently to generate `X_all_train` and `X_all_test`.

Nevertheless, the target variable `income` has been a binary categorical variable with the levels "`<=50K`" and "`>50K`". Hence, there must be two dummy variables in the DataFrame, one for each of the categories. For model construction, however, we only need one of them, and the decision here favours the dummy variable "`income_>50K`" where 1 represents observation with income more than 50,000 USD and 0 the opposite. Through this value assignment, the natural order of the income can be reflected by the order of values in this dummy variable, and we can avoid confusion when interpreting the modelling results.



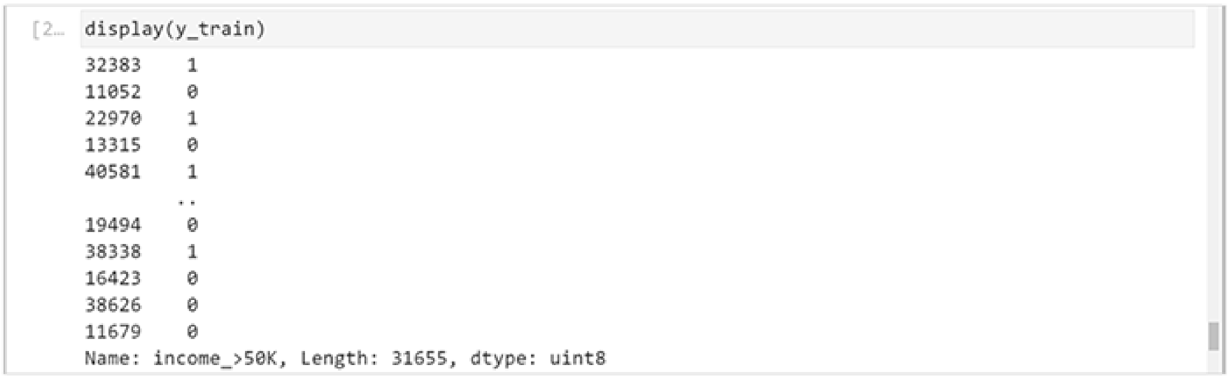
```
[2… y_dummyvar = ["income_<=50K", "income_>50K"]
    y_train = DF_model_train[y_dummyvar[1]]
    y_test = DF_model_test[y_dummyvar[1]]
    X_all_train = DF_model_train.drop(y_dummyvar, axis = 1)
    X_all_test = DF_model_test.drop(y_dummyvar, axis = 1)
```

Figure 5.19 Slicing Training and Testing Datasets for the Independent and the Target Variables

In the first line, we create a list called `y_dummyvar` with the names of the two dummy variables in it. We can use this list to remove the corresponding columns in `X_all_train` and `X_all_test`. For the construction of `y_train` and `y_test`, the column with the label that matches the second item in `y_dummyvar` will be selected from the training and testing DataFrames.

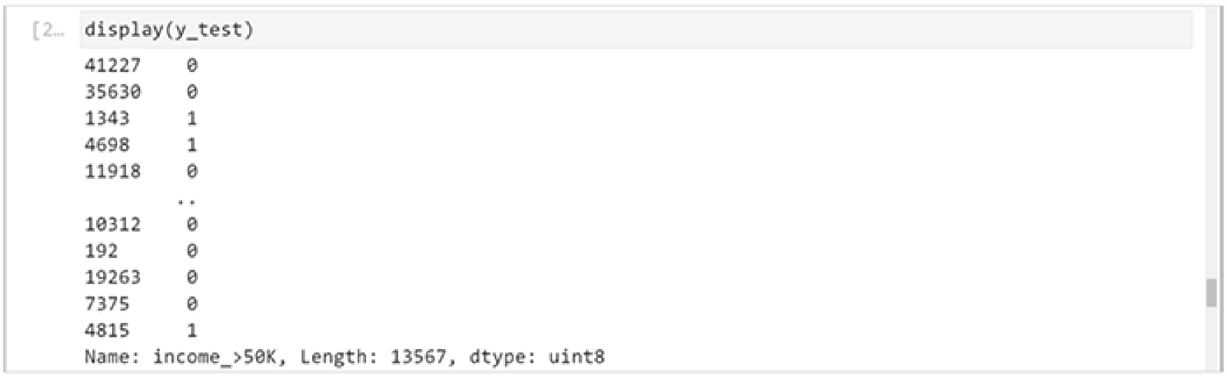
Note that both `X_all_train` and `X_all_test` still contain the original numeric variables as well as their normalised counterparts. We keep both sets purposely so that we can respond to the requirements of the different algorithms flexibly by choosing either set of them. This is also the reason why we have created two lists named `numvar_list` and `numnormvar_list` in the previous sections so that we can select the columns directly from the DataFrames in the future.

From Figure 5.20 and Figure 5.21, we can see that `y_train` and `y_test` have been converted to two pandas Series with 31,655 and 13,567 elements, respectively. Figure 5.22 and Figure 5.23 show the training and testing DataFrames of the independent variables `X_all_train` and `X_all_test`.



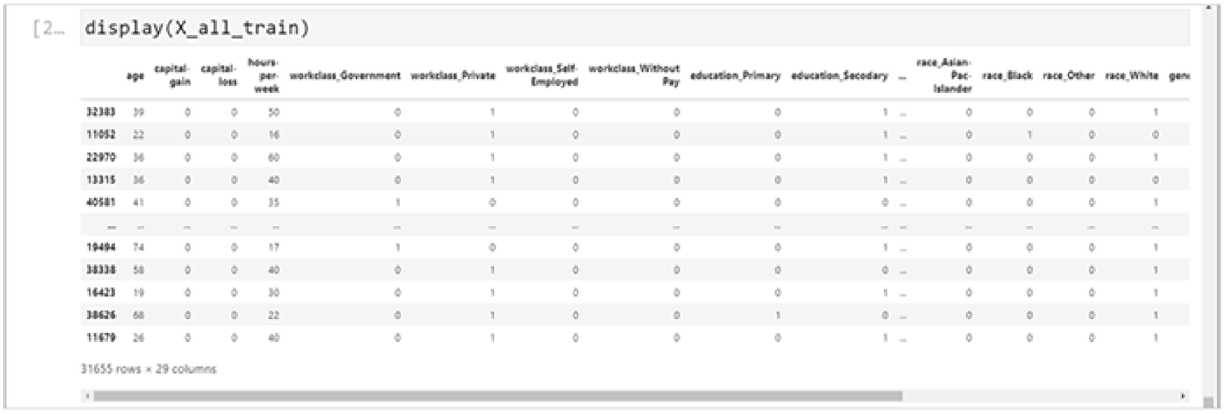
```
[2… display(y_train)
    32383    1
    11052    0
    22970    1
    13315    0
    40581    1
            ..
    19494    0
    38338    1
    16423    0
    38626    0
    11679    0
    Name: income_>50K, Length: 31655, dtype: uint8
```

Figure 5.20 Training Dataset for the Target Variable



```
[2… display(y_test)
41227    0
35630    0
1343     1
4698     1
11918    0
        ..
10312    0
192      0
19263    0
7375     0
4815     1
Name: income_>50K, Length: 13567, dtype: uint8
```

Figure 5.21 Testing Dataset for the Target Variable

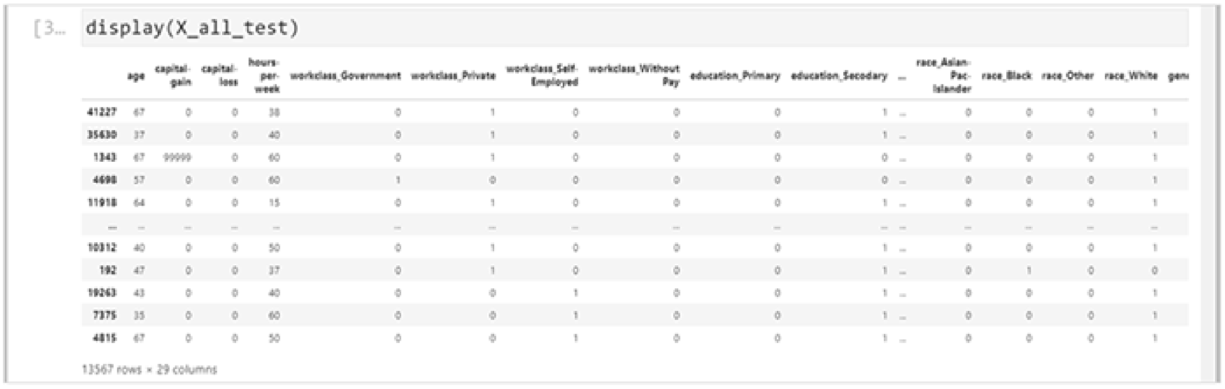


```
[2… display(X_all_train)
```

	age	capital-gain	capital-loss	hours-per-week	workclass_Government	workclass_Private	workclass_Self-Employed	workclass_Without Pay	education_Primary	education_Secodary	...	race_Asian-Pac-Islander	race_Black	race_Other	race_White	gen
32383	39	0	0	50	0	1	0	0	0	1	...	0	0	0	1	
11052	22	0	0	16	0	1	0	0	0	1	...	0	1	0	0	
22970	36	0	0	60	0	1	0	0	0	1	...	0	0	0	1	
13315	36	0	0	40	0	1	0	0	0	1	...	0	0	0	0	
40581	41	0	0	35	1	0	0	0	0	0	...	0	0	0	1	
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	
19494	74	0	0	17	1	0	0	0	0	1	...	0	0	0	1	
38338	58	0	0	40	0	1	0	0	0	0	...	0	0	0	1	
16423	19	0	0	30	0	1	0	0	0	1	...	0	0	0	1	
38626	68	0	0	22	0	1	0	0	1	0	...	0	0	0	1	
11679	26	0	0	40	0	1	0	0	0	1	...	0	0	0	1	

31655 rows × 29 columns

Figure 5.22 Training Dataset for the Independent Variables



```
[3… display(X_all_test)
```

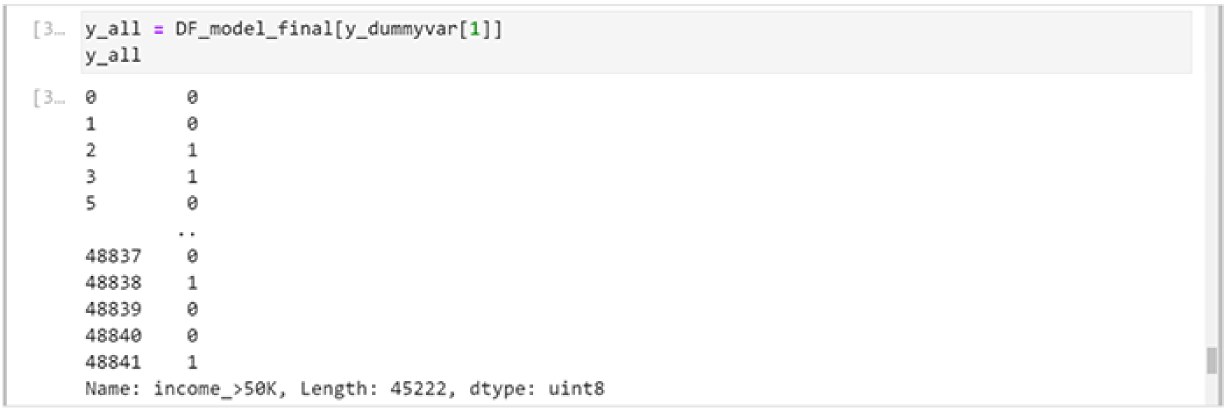
	age	capital-gain	capital-loss	hours-per-week	workclass_Government	workclass_Private	workclass_Self-Employed	workclass_Without Pay	education_Primary	education_Secodary	...	race_Asian-Pac-Islander	race_Black	race_Other	race_White	gen
41227	67	0	0	38	0	1	0	0	0	1	...	0	0	0	1	
35630	37	0	0	40	0	1	0	0	0	1	...	0	0	0	1	
1343	67	99999	0	60	0	1	0	0	0	0	...	0	0	0	1	
4698	57	0	0	60	1	0	0	0	0	0	...	0	0	0	1	
11918	64	0	0	15	0	1	0	0	0	1	...	0	0	0	1	
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	
10312	40	0	0	50	0	1	0	0	0	1	...	0	0	0	1	
192	47	0	0	37	0	1	0	0	0	1	...	0	1	0	0	
19263	43	0	0	40	0	0	1	0	0	1	...	0	0	0	1	
7375	35	0	0	60	0	0	1	0	0	1	...	0	0	0	1	
4815	67	0	0	50	0	0	1	0	0	1	...	0	0	0	1	

13567 rows × 29 columns

Figure 5.23 Testing Dataset for the Independent Variables

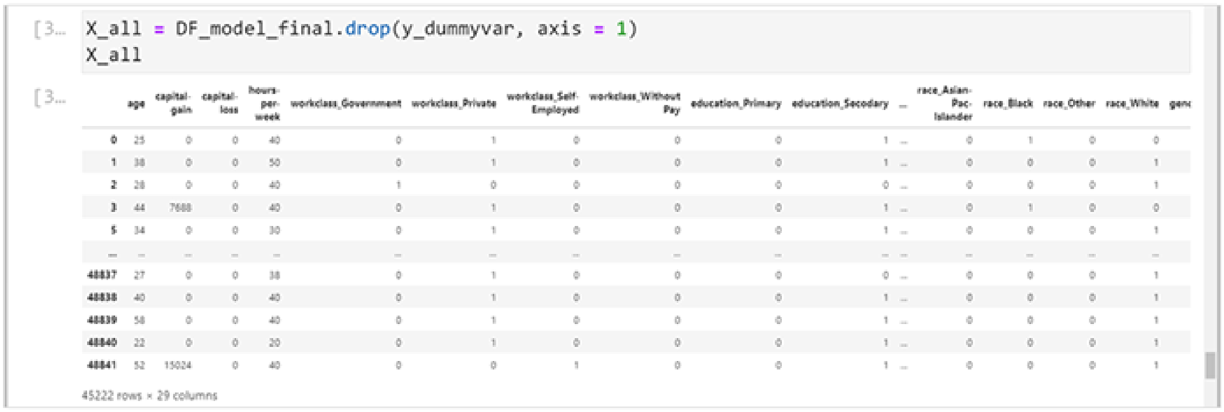
When applying unsupervised machine learning algorithms where we do not need to partition our DataFrame into training and testing datasets, we can split the dependent variables and the input variables directly from the original DataFrame

`DF_model_final`. The mechanism is just the same as Figure 5.19. The corresponding syntaxes and outputs are given in Figure 5.24 and Figure 5.25.



```
[3… y_all = DF_model_final[y_dummyvar[1]]
    y_all

[3… 0        0
    1        0
    2        1
    3        1
    5        0
            ..
    48837    0
    48838    1
    48839    0
    48840    0
    48841    1
    Name: income_>50K, Length: 45222, dtype: uint8
```

Figure 5.24 Dataset for the Dependent Variable



```
[3… X_all = DF_model_final.drop(y_dummyvar, axis = 1)
    X_all
```

[3…

	age	capital-gain	capital-loss	hours-per-week	workclass_Government	workclass_Private	workclass_Self-Employed	workclass_Without Pay	education_Primary	education_Secodary	...	race_Asian-Pac-Islander	race_Black	race_Other	race_White	gen
0	25	0	0	40	0	1	0	0	0	1	...	0	1	0	0	
1	38	0	0	50	0	1	0	0	0	1	...	0	0	0	1	
2	28	0	0	40	1	0	0	0	0	0	...	0	0	0	1	
3	44	7688	0	40	0	1	0	0	0	1	...	0	1	0	0	
5	34	0	0	30	0	1	0	0	0	1	...	0	0	0	1	
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	
48837	27	0	0	38	0	1	0	0	0	0	...	0	0	0	1	
48838	40	0	0	40	0	1	0	0	0	1	...	0	0	0	1	
48839	58	0	0	40	0	1	0	0	0	1	...	0	0	0	1	
48840	22	0	0	20	0	1	0	0	0	1	...	0	0	0	1	
48841	52	15024	0	40	0	0	1	0	0	1	...	0	0	0	1	

45222 rows × 29 columns

Figure 5.25 Dataset for the Independent Variables

Chapter 2: Clustering

2.1 Introduction of K-Means Clustering

Lesson Recording

Introduction and Fitting of K-Means Clustering by scikit-learn

Clustering is a multivariate analytics technique to group “similar” observations into finite number of disjoint clusters. One of the most popular clustering algorithms is the K-Means method. This technique is very efficient in clustering large data sets. The algorithm here is to split the data into K groups with equal variance by minimising the variation within the cluster. This variation is called the inertia or within-cluster sum-of-squares. In other words, “members” in the same cluster should be as “similar” as possible, while observations from different clusters should be most distinguishable.

Different from some other clustering algorithms in which the number of clusters will only emerge during the grouping process, the K-Means method requires the number of clusters to be specified before the algorithm starts. The clusters are characterised by their centroids, which can be interpreted as the centre of an area in a two-dimensional space, and are hence the average of all the observations within a cluster. As the name of the algorithm suggests, there should be K different means (centroids) and they should be explored during the clustering process.

The process of K-Means clustering can be described in five main steps:

1. K observations are randomly selected as initial cluster centroids where K is a pre-defined positive integer.
2. Compute the distance of each object to the centroids.
3. Based on the distance computed, each object is assigned to the nearest centroid. Objects assigned to the same centroid form a cluster. There will be K different clusters.
4. For each cluster, recompute the centroid using the objects assigned to the cluster. The iteration starts again from Step 2.
5. The iteration stops when the centroids remain unchanged or a specified number of iterations has been performed.

Note that the distance mentioned in 2) refers to the Euclidean distance in general. The Euclidean distance between an object and a cluster centroid is measured by the sum of the squared differences between the values of some selected clustering criteria, which are usually some input variables of the object, and the values of the same clustering criteria of the centroid.

Subsequent to the clustering process, it is important to make sure that the resulting clusters really create some insights. To interpret the clusters, the characteristics of each cluster should be explored by looking at the summary statistics (e.g. mean, min, max) of the clustering criteria. A good clustering solution should allow us to describe the profile of each cluster clearly.

In addition, there are objective measures for evaluating the quality of clustering solutions: cohesion, separation and parsimony. The cohesion measures the similarity of the objects in a cluster. This value should be small because these objects should be similar. The separation, on the other hand, measures how dissimilar the clusters are, and this value should be high. Here, we can apply the Silhouette coefficient since it combines both the cohesion and the separation. Briefly speaking, the Silhouette coefficient is a value between -1 and 1 that measures the relationship between the intra-cluster distances and nearest cluster distances. The mean of the individual Silhouette coefficients will be computed for every clustering solution for evaluation. A high and positive average Silhouette coefficient suggests appropriate and useful clustering solution. On the contrary, negative Silhouette coefficient indicates a rather undesirable clustering result.

Furthermore, parsimony is another important criterion in clustering. As a result, we prefer smaller number of clusters if the quality of the corresponding clustering solution is satisfactory. Nevertheless, the number of clustering criteria should also be parsimonious, so that the clustering solution can be interpreted conveniently.

2.2 Fitting K-Means Clustering by Scikit-Learn

In scikit-learn, all algorithms are controlled and executed by the so-called estimator. We can adjust our parameters for the modelling process in the syntax of the estimator declaration. In K-Means Clustering, the estimator is called `KMeans`. And it can be imported from the cluster module of the sklearn package.

```
from sklearn.cluster import KMeans
```

First, we need to initiate the `KMeans` estimator and adjust the estimation parameters according to our needs.

```
km_Object = sklearn.cluster.KMeans(n_clusters = 8, init =
             "k-means++", n_init = 10, max_iter = 300,
             tol = 0.0001, precompute_distances = "auto",
             random_state = None)
```

The following table provides description and explanation of the parameters.

Table 5.2 Parameters of the KMeans Estimator

Parameter	**Value Type**	**Description**
n_clusters *(Default: 8)*	integer	The number of clusters to form as well as the number of centroids to generate.
init *(Default: "k-means++")*	"k-means++", "random", callable, array-like of shape	Method for initialisation: "k-means++": selects initial cluster centres for k-mean clustering in a smart way to speed up convergence. "random": choose n_clusters observations at random from the data as initial centroids. Array: An array with number of rows equal to the n_clusters and number of columns equal to the number of variables that give the initial centroids. Callable: It should take arguments X, n_clusters and a random state and return an initialisation.
n_init *(Default: 10)*	integer	Number of times the k-means algorithm will be run with different centroid seeds. The final results will be the best output of n_init consecutive runs in terms of inertia.
max_iter *(Default: 300)*	integer	Maximum number of iterations of the k-means algorithm for a single run.

Parameter	Value Type	Description
`tol` *(Default: 1e-4)*	float	Relative tolerance of the difference in the cluster centres of two consecutive iterations to declare convergence.
`precompute_distances` *(Default: `"auto"`)*	`"auto"`, `True`, `False`	Precompute distances (faster but takes more memory). `"auto"`: do not precompute distances if `n_samples * n_clusters` > 12 million. This corresponds to about 100MB overhead per job using double precision. `True`: always precompute distances. `False`: never precompute distances.
random_state *(Default: `None`)*	integer, RandomState instance, `None`	Determines random number generation for centroid initialisation. Use an integer to make the randomness deterministic.

(Source: https://scikit-learn.org/stable/modules/generated/sklearn.cluster.KMeans.html)

Here are some explanations regarding some items mentioned in Table 5.2:

- A callable is a program part that can be called such as a user-defined function or a built-in method.
- Centroid seeds refer to the initial cluster centres.
- Inertia measures the within-cluster sum-of-squares and should be minimised in the clustering process.

Next, we can apply the `KMeans` estimator on a prepared DataFrame.

```
km_fit_Object = km_Object.fit(X, sample_weight = None)
```

The parameter `X` is a prepared DataFrame based on which the clusters are constructed. With `sample_weight` we can pre-specify the weights for each observation in `X`. If it is set to `None`, which is also the default here, all observations will be assigned equal weight.

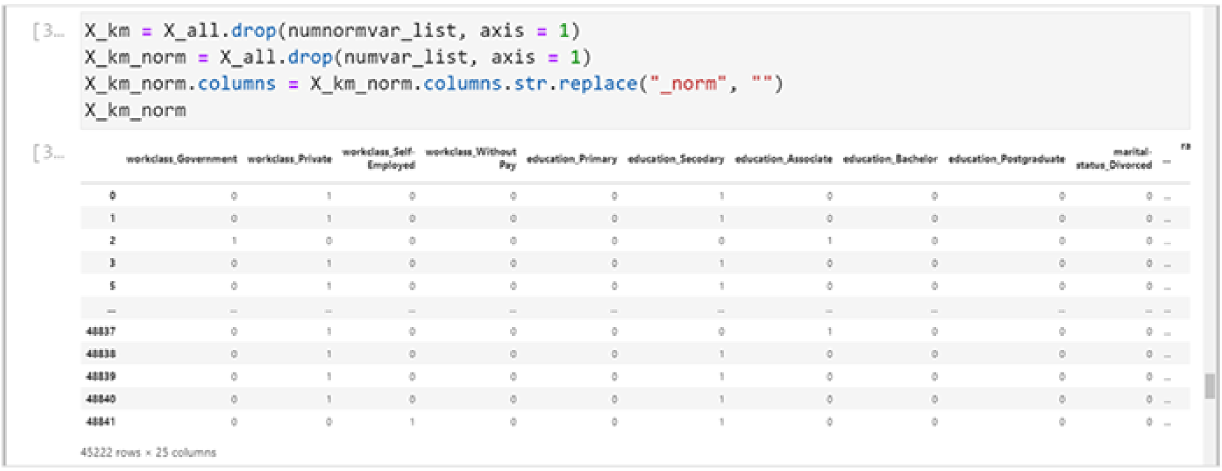
The fitted estimator of the K-Means algorithm is saved in `km_fit_Object`. To obtain and view the results of the estimation, we still need to request scikit-learn to predict the cluster classification for each observation in the DataFrame `X`.

```
km_pred_Object = km_Object.fit_predict(X, sample_weight =
                 None)
```

The parameters here are the same as the `.fit()` function. The parameter `X` contains the data for which the cluster prediction will be calculated. The pre-specified individual weights in `sample_weight` will be assigned to all observations in the dataset. The output object here is an n-dimensional array of length `n_samples`, i.e., the number of observations in the dataset. The items in the array are indices of the cluster that each sample belongs to.

As mentioned in Chapter 2.1, the number of clusters, K, must be specified before the clustering algorithm starts. One way to determine the optimal value of K is the elbow method. Elbow method is a popular technique that uses the inertia as the measurement to compare the distortions in some clustering solutions with different K. The distortion is the sum of squared distances of each data point to the centroids. The plot of distortions which looks like an arm will then be generated. The best value of K can be found at the "elbow", the inflection point on the curve. To determine the inertia of a clustering solution, we can apply the `.inertia_` method on a `KMeans` estimator.

Example (Cont'd): Since K-Means clustering is an unsupervised machine learning algorithm, we do not need to split our data into a training and a testing dataset. We can use the entire available DataFrame for clustering purpose.



```
[3… X_km = X_all.drop(numnormvar_list, axis = 1)
    X_km_norm = X_all.drop(numvar_list, axis = 1)
    X_km_norm.columns = X_km_norm.columns.str.replace("_norm", "")
    X_km_norm
```

[3…

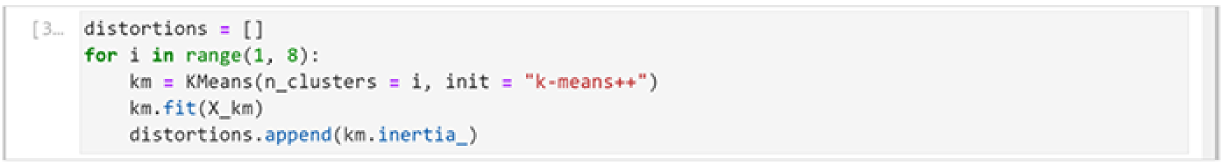
	workclass_Government	workclass_Private	workclass_Self-Employed	workclass_Without Pay	education_Primary	education_Secodary	education_Associate	education_Bachelor	education_Postgraduate	marital-status_Divorced	...
0	0	1	0	0	0	1	0	0	0	0	...
1	0	1	0	0	0	1	0	0	0	0	...
2	1	0	0	0	0	0	1	0	0	0	...
3	0	1	0	0	0	1	0	0	0	0	...
5	0	1	0	0	0	1	0	0	0	0	...
...	...	...	...	...	...	...	...	...	...	...	...
48837	0	1	0	0	0	0	1	0	0	0	...
48838	0	1	0	0	0	1	0	0	0	0	...
48839	0	1	0	0	0	1	0	0	0	0	...
48840	0	1	0	0	0	1	0	0	0	0	...
48841	0	0	1	0	0	1	0	0	0	0	...

45222 rows × 25 columns

Figure 5.26 Create a DataFrame with Only Normalised Input Variables

Nevertheless, as mentioned in Chapter 1.2.6., it is more sensible to include normalised data to build the clusters since numeric variables with extreme value ranges such as `capital-gain` or `capital-loss` have to be scaled. In the first line, `X_km` is the DataFrame with no normalised variables and is used for the elbow-test to determine the optimal number of clusters. The name of all the normalised variables are stored in the list `numnormvar_list` (see Figure 5.15). In the second line, `X_km_norm` is the DataFrame with no non-normalised variables and will be used for the clustering process. The name of all the numeric variables before normalisation is stored in numvar_list (see Figure 5.13). In `X_km_norm`, we rename the normalised variables back to their original variable names by removing their suffix "_norm" in order to simplify their labels for later output.

To find out the best K, the number of clusters, we can conduct an elbow test. Here, we will compute the inertia of the K-Means clustering solutions that contains 1 to 7 clusters and compare their distortions.

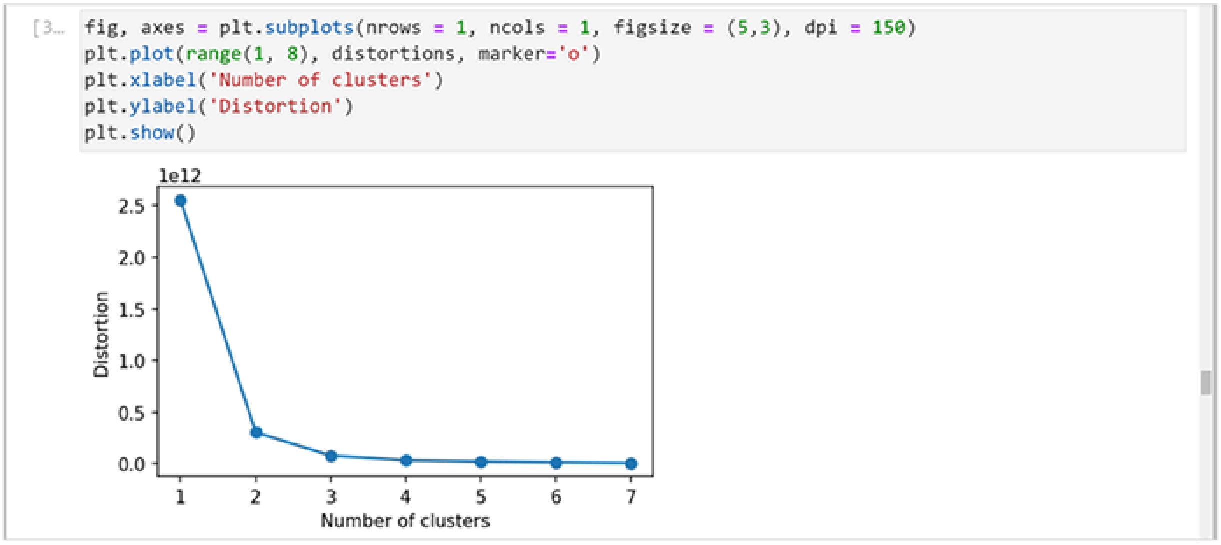


```
[3… distortions = []
    for i in range(1, 8):
        km = KMeans(n_clusters = i, init = "k-means++")
        km.fit(X_km)
        distortions.append(km.inertia_)
```

Figure 5.27 Calculating Inertia for the Elbow Method

We use a `for`-loop here to run through all clustering solutions with 1 to 7 clusters. The parameter in the KMeans estimator is kept as simple as possible. We only set the number of clusters and instruct Python to select initial cluster centres for k-mean clustering in a smart way to speed up convergence by placing `init = "k-means++"`. After fitting the K-Means clusters on our data, we store the corresponding inertia in the list named `distortions` for later use.

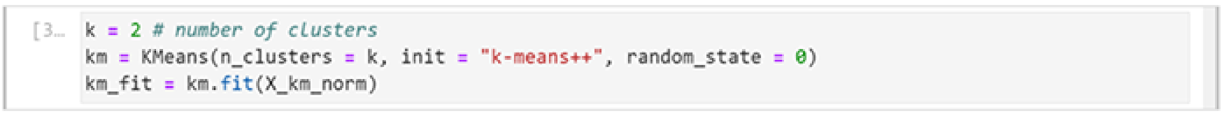
After the inertia are calculated, we can plot them for the Elbow method.



```
[3… fig, axes = plt.subplots(nrows = 1, ncols = 1, figsize = (5,3), dpi = 150)
    plt.plot(range(1, 8), distortions, marker='o')
    plt.xlabel('Number of clusters')
    plt.ylabel('Distortion')
    plt.show()
```

Figure 5.28 Elbow Method to Determine the Optimal Number of Clusters

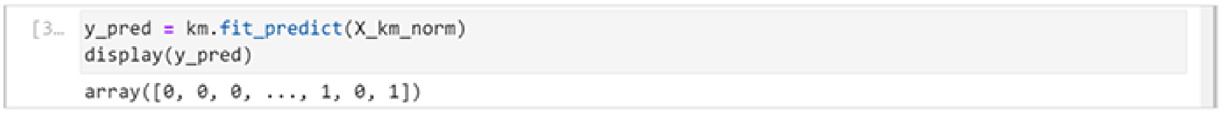
In the first line of Figure 5.28, we use the matplotlib options to set the size and resolution of the chart. In the second line, we put the number of clusters in the clustering solutions on the x-axis and the inertia on the y-axis. From the shape of the graph, the elbow can be found at `K = 2`. As a result, we will use a 2-cluster solution in the following.



```
[3… k = 2 # number of clusters
    km = KMeans(n_clusters = k, init = "k-means++", random_state = 0)
    km_fit = km.fit(X_km_norm)
```

Figure 5.29 Fitting K-Means Clustering

The code in Figure 5.29 is basically identical to those in Figure 5.27 but without the loop. Here, we fix our number of clusters to 2 and the `random_state` to 0, which enables us to reproduce the same clustering results in the future.



```
[3… y_pred = km.fit_predict(X_km_norm)
    display(y_pred)

    array([0, 0, 0, ..., 1, 0, 1])
```

Figure 5.30 Predicting Classification of the Data

After creating the K-Means clusters based on our normalised numeric variables and the dummy variables of our categorical variables, we save the predicted cluster index of each observation in the array object named `y_pred`. From the output, we can see that the cluster indices are 0 or 1.

Read

Refer to the link below for more details and examples on the `KMeans` estimator, `fit()`, and `fit_predict()` functions of the `cluster` module in the scikit-learn package:

https://scikit-learn.org/stable/modules/generated/sklearn.cluster.KMeans.html

2.3 Model Exploration and Evaluation

Lesson Recording

Explore & Evaluate K-Means Clustering Models by scikit-learn

As mentioned in the previous section, the characteristics of the clusters should be explored and interpreted. We can examine this by looking at some statistics such as mean, min, max, etc. of the clustering criteria based on which the clusters have been constructed. For this purpose, we can cross-tabulate the clustering criteria and the cluster index to understand the features of the clusters.

If a clustering criteria variable is categorical, our focus of interpretation will be on the proportional distribution of the clusters in each category. With the `crosstab()` function, which we have briefly introduced in Chapter 1.2.3, we can easily create a cross-table to fulfil our purpose.

```
pd.crosstab(index = criteria_var, columns = cluster_index,
            normalize = "index", margin = True)
```

The object assigned to the first parameter is the row variable of the cross-tabulation. In most of the cases, one of the clustering criteria is placed here since the number of categories could be rather large, and it is more convenient to have them listed in rows rather than in columns. If multiple names are passed to this parameter, these variables will be tabulated in hierarchy. The second parameter controls the column variable. Our suggestion is to place the cluster index here since the number of clusters is usually limited. The parameter `normalize` can take the values `"all"`, `"index"`, `"columns"`, `{0, 1}`, or `{True, False}` where the default value is `False`, or `0` equivalently. If `normalize = True` or

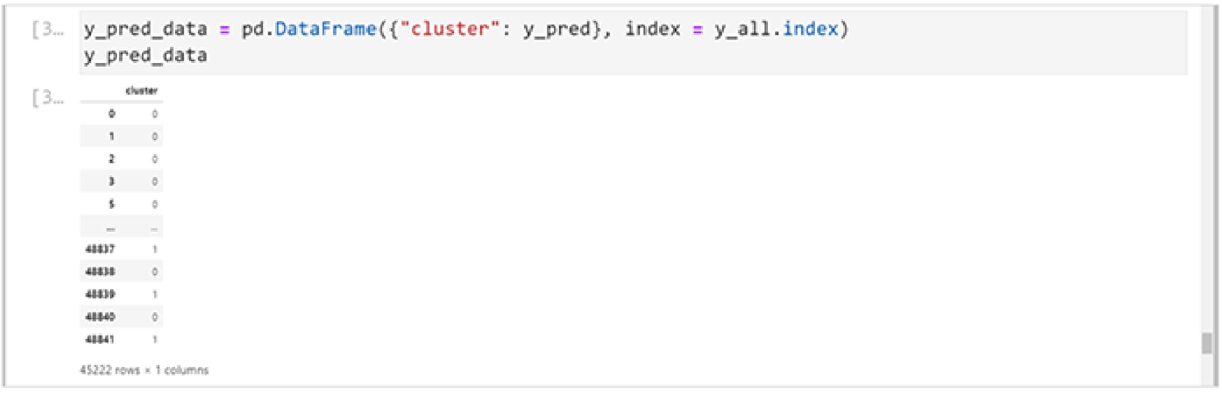
`1`, Python will return the table percentage of each cell, while the counts of each cell will be shown if `normalize` is `False` or `0`. If `normalize = "index"`, the row proportion of the cell will be calculated, while the column percentage will be provided if `normalize = "column"`. If `margin = True`, the marginal frequency of the axis specified in the `normalize` will be listed out as well.

If a clustering criterion were numeric, the `crosstab()` function would not be a good choice since it would take every unique value in it as a separate category. In this case, we would rather let Python calculate some statistics of the clustering criteria for each cluster. And the `.groupby()` method of the pandas package, which has already been introduced in Chapter 5.3 of Study Unit 4, would become applicable in our Python program once again.

```
DF[[criteria_var, cluster_index]]
      .groupby(by = [cluster_index]).anymethod().transpose()
```

The above syntax is specifically adjusted for clustering interpretation. First, we merge the selected clustering criteria variables from the original DataFrame with the cluster classification variable. The resulting DataFrame will be grouped by the cluster indices, and the method to compute the statistics of interest is appended to it. Note that the variable used in the .groupby() method, which is the cluster index in this case, will be displayed as rows. But we can transpose the result of the `.groupby()` method to swap the rows and columns. As a result, we put the cluster indices as columns in the same way `crosstab()` does. But it is more a step to standardise the presentation rather than an analytics requirement and hence optional for us to integrate it in our syntax or not.

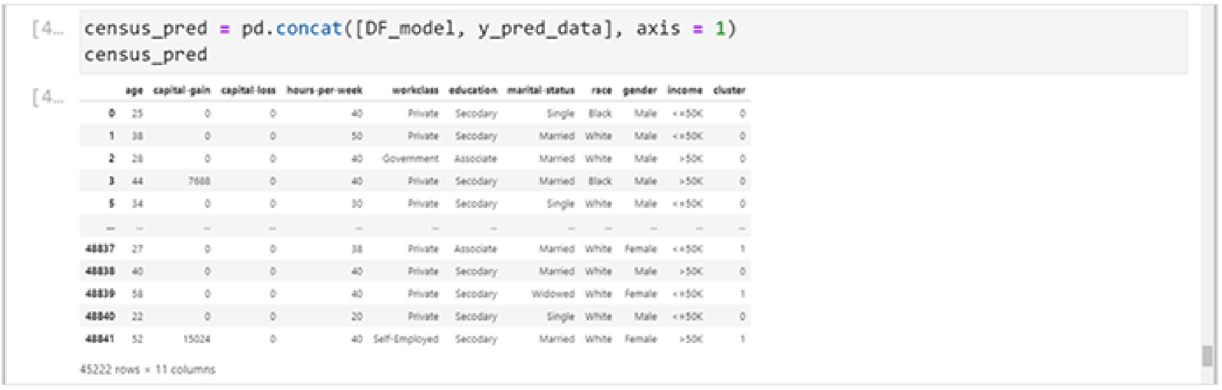
Example (Cont'd): First, we convert the cluster index array to become a pandas DataFrame and name the variable "`cluster`".



```
[3… y_pred_data = pd.DataFrame({"cluster": y_pred}, index = y_all.index)
    y_pred_data

[3…        cluster
        0        0
        1        0
        2        0
        3        0
        5        0
      ...      ...
    48837        1
    48838        0
    48839        1
    48840        0
    48841        1

    45222 rows × 1 columns
```

Figure 5.31 Converting the Cluster Index Array to pandas DataFrame

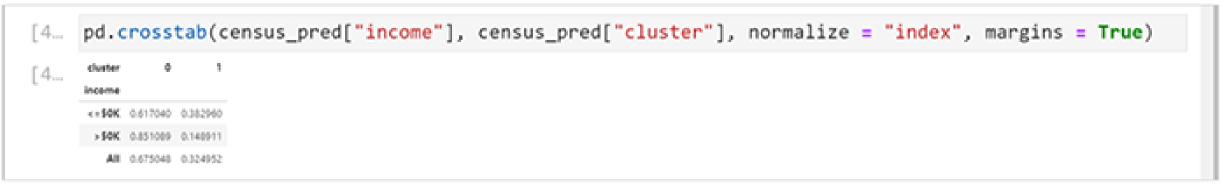
Subsequently, we concatenate the original DataFrame (before the dummy and normalised variables are created) and the cluster index variable.



```
[4… census_pred = pd.concat([DF_model, y_pred_data], axis = 1)
    census_pred

[4…       age capital-gain capital-loss hours-per-week     workclass education marital-status  race gender income cluster
        0  25            0            0             40       Private  Secodary         Single Black   Male  <=50K       0
        1  38            0            0             50       Private  Secodary        Married White   Male  <=50K       0
        2  28            0            0             40    Government Associate        Married White   Male   >50K       0
        3  44         7688            0             40       Private  Secodary        Married Black   Male   >50K       0
        5  34            0            0             30       Private  Secodary         Single White   Male  <=50K       0
      ... ...          ...          ...            ...           ...       ...            ...   ...    ...    ...     ...
    48837  27            0            0             38       Private Associate        Married White Female  <=50K       1
    48838  40            0            0             40       Private  Secodary        Married White   Male   >50K       0
    48839  58            0            0             40       Private  Secodary        Widowed White Female  <=50K       1
    48840  22            0            0             20       Private  Secodary         Single White   Male  <=50K       0
    48841  52        15024            0             40 Self-Employed  Secodary        Married White Female   >50K       1

    45222 rows × 11 columns
```

Figure 5.32 Merging the Original DataFrame with the Cluster Index

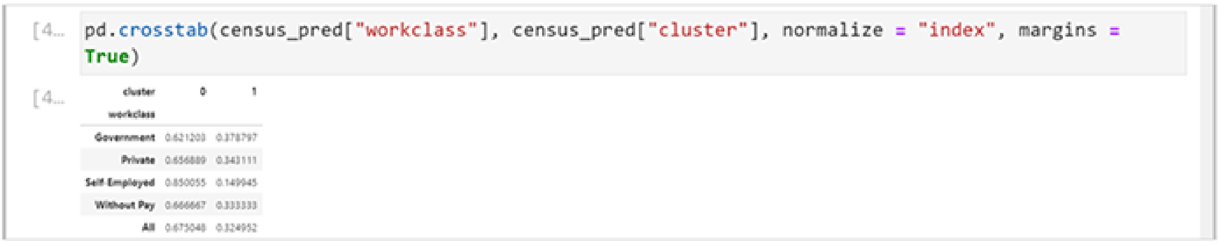
With this DataFrame, we can create our cross tables with the appropriate labels.



```
[4… pd.crosstab(census_pred["income"], census_pred["cluster"], normalize = "index", margins = True)

[4… cluster         0         1
     income
      <=50K  0.617040  0.382960
       >50K  0.851089  0.148911
        All  0.675048  0.324952
```

Figure 5.33 Cross-Tabulation of `income` and the Predicated Classification

From the above table, we can see that the majority of the observations with income more than 50,000 USD are in cluster 0. Though this cluster also contains 62% of the low-income group, the proportion of the same income group is rather high in cluster 1, namely 38%. The first impression is that the observations in cluster 0 are financially more well-off.

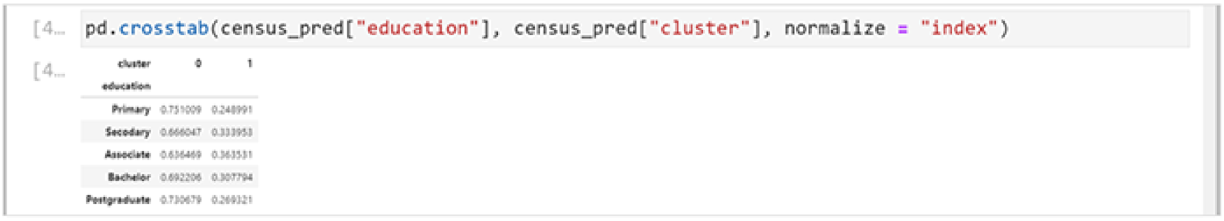


```
[4… pd.crosstab(census_pred["workclass"], census_pred["cluster"], normalize = "index", margins =
    True)
```

```
[4…       cluster         0         1
        workclass
       Government  0.621203  0.378797
          Private  0.656889  0.343111
    Self-Employed  0.850055  0.149945
      Without Pay  0.666667  0.333333
              All  0.675048  0.324952
```

Figure 5.34 Cross-Tabulation of `workclass` and the Predicated Classification

Figure 5.34 shows the cross-tabulation of `workclass` and the cluster index. Here, we can see that while cluster 0 contains most of the self-employed individuals (85% vs. 15%), the distributions of the other work classes correspond roughly to the distribution of the clusters in the entire sample (68% vs. 32%).

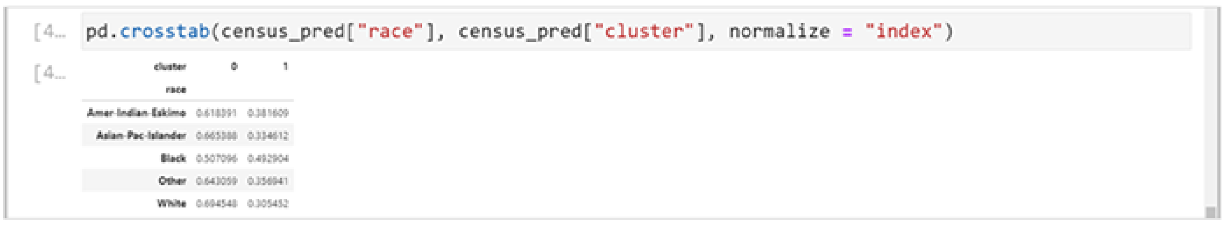


```
[4… pd.crosstab(census_pred["education"], census_pred["cluster"], normalize = "index")
```

```
[4…       cluster         0         1
        education
          Primary  0.751009  0.248991
         Secodary  0.666047  0.333953
        Associate  0.636469  0.363531
         Bachelor  0.692206  0.307794
     Postgraduate  0.730679  0.269321
```

Figure 5.35 Cross-Tabulation of `education` and the Predicated Classification

The cross-tabulation of `education` and the cluster index does not provide a conclusive relationship between them. While individuals with primary education level are over proportionally represented in cluster 0, postgraduates share a similar proportion in this cluster. As a result, the clusters do not differentiate the individual education level at all.

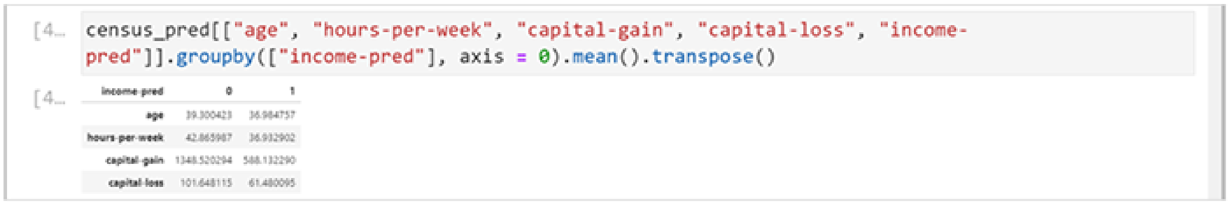


```
[4… pd.crosstab(census_pred["race"], census_pred["cluster"], normalize = "index")
```

```
[4…             cluster         0         1
                   race
     Amer-Indian-Eskimo  0.618391  0.381609
     Asian-Pac-Islander  0.665388  0.334612
                  Black  0.507096  0.492904
                  Other  0.643059  0.356941
                  White  0.694548  0.305452
```

Figure 5.36 Cross-Tabulation of `race` and the Predicated Classification

From this cross-tabulation, Afro-Americans are over proportionally represented in cluster 1 than other ethnic groups.



```
[4… census_pred[["age", "hours-per-week", "capital-gain", "capital-loss", "income-
    pred"]].groupby(["income-pred"], axis = 0).mean().transpose()
```

```
[4…    income-pred            0           1
               age    39.300423   36.984757
    hours-per-week    42.865987   36.932902
      capital-gain  1348.520294  588.132290
      capital-loss   101.648115   61.480095
```

Figure 5.37 Cross-Tabulation of Numeric Variables and the Predicated Classification

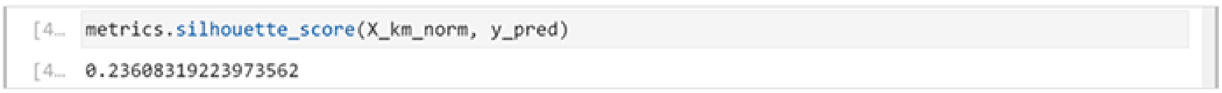
From Figure 5.37, we can see that individuals in cluster 0 work averagely 6 hours more in a week and their ratio of capital-gain to capital-loss is much higher than those in cluster 1. Their average age, however, does not differ significantly in both clusters.

In Chapter 2.1, we introduced the Silhouette coefficient as a measure to evaluate the cohesion and separation of a clustering solution. In scikit-learn, we can apply the following syntax to calculate it.

```
metrics.silhouette_score(criteria_var, cluster_index)
```

Note that the `silhouette_score()` function is from `metrics` and not the `KMeans` module.

Example (Cont'd): To calculate the Silhouette coefficient, we need the clustering criteria array that is stored in the DataFrame `X_km_norm` and the cluster index array named `y_pred`.



```
[4… metrics.silhouette_score(X_km_norm, y_pred)
[4… 0.23608319223973562
```

Figure 5.38 Calculating the Silhouette Coefficient

The closer the Silhouette coefficient is to 1, the better is the cohesion and separation of the cluster. Here, a coefficient of 0.24 is indeed not very promising. In other words, the similarity within the cluster and the dissimilarity among the clusters are not very clear in this clustering solution. This confirms the findings resulting from the previous cross-tabulations in which the clusters do not provide a clear differentiation of some of the clustering criteria as well.

Another possibility to understand a clustering solution is the graphical approach. The idea of this approach is to plot all the data points with their cluster classification in a two-dimensional scatter plot. This approach would be rather straightforward if only one or two input variables have been used as clustering criteria. In the multivariate case, we need to reduce the dimensionality of all the input variables down to two before plotting.

One of the most common methods to reduce the dimensions of a high-dimensional array (or matrix) is the Principal Component Analysis (PCA). The idea of the PCA is to project each data point onto the first few principal components which contain the majority of the variation in the data. The loss of information caused by the omission of the remaining components is then insignificant and the data dimension is reduced to the number of principal components. In our case, we may only be interested in the first two components to span the space for our scatter plot.

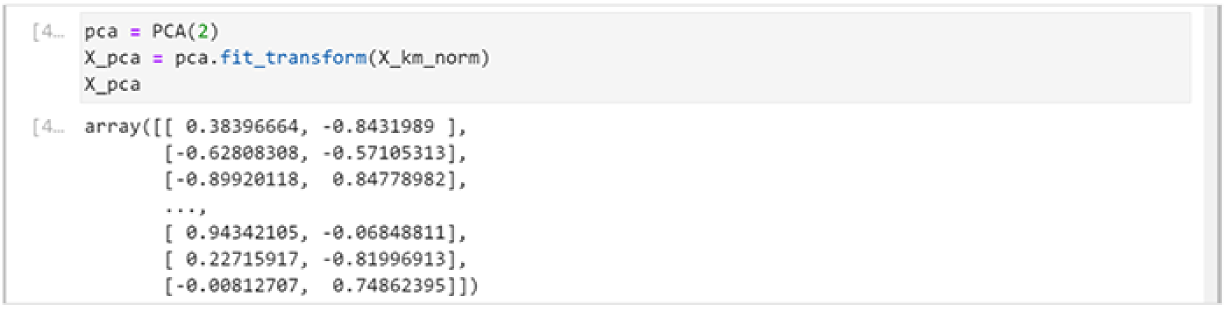
```
pca_Object = sklearn.decomposition.PCA(n_components)
```

Note that `PCA()` is an estimator from the `decomposition` module. With the above syntax, we can specify the number of components we would like the result to contain. To reduce the dimension an array by `PCA()`, we need to execute the following syntax.

```
pca_fit_Object =
 sklearn.decomposition.fit_transform(criteria_var)
```

Here, we need to use the `.fit_transformation()` instead of the `.fit()` function since we are interested in the transformed values of all the criteria variables. The returned object is an n-dimensional NumPy array with the transformed data. Hence, the array has 2 columns, and its number of rows corresponds to the number of observations in the DataFrame that has been passed to the `.fit_transformation()` function in the first place.

Example (Cont'd): In the first step, we reduce the dimensionality of the clustering criteria array `X_km_norm` to 2 components.

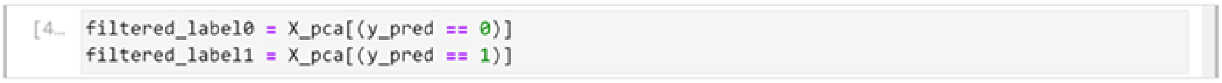


```
[4… pca = PCA(2)
    X_pca = pca.fit_transform(X_km_norm)
    X_pca

[4… array([[ 0.38396664, -0.8431989 ],
           [-0.62808308, -0.57105313],
           [-0.89920118,  0.84778982],
           ...,
           [ 0.94342105, -0.06848811],
           [ 0.22715917, -0.81996913],
           [-0.00812707,  0.74862395]])
```

Figure 5.39 Reducing the Dimensionality of the Input Variable DataFrame

The resulting values in the array `X_pca` are the coordinates of each data point in the scatter plot.

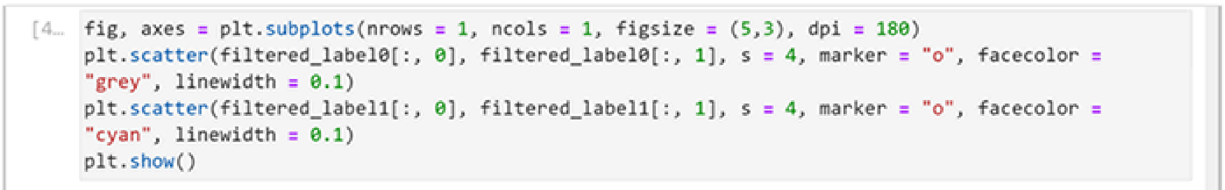
In the next step, we separate the coordinates of observations of cluster 0 from those of cluster 1 since we would like to plot them in different colours in the final chart.



```
filtered_label0 = X_pca[(y_pred == 0)]
filtered_label1 = X_pca[(y_pred == 1)]
```

Figure 5.40 Selecting PCA Data Based on the Predicted Classification

To plot the data points of different clusters in different colours, we need two `plt.scatter` command since we have to fix the `facecolour` parameter in every line. The values on the x-axis are the values stored in the first column of the subset arrays of `X_pca` called `filtered_label0` and `filtered_label1`. The values in their second columns are the values on the y-axis.



```
fig, axes = plt.subplots(nrows = 1, ncols = 1, figsize = (5,3), dpi = 180)
plt.scatter(filtered_label0[:, 0], filtered_label0[:, 1], s = 4, marker = "o", facecolor =
"grey", linewidth = 0.1)
plt.scatter(filtered_label1[:, 0], filtered_label1[:, 1], s = 4, marker = "o", facecolor =
"cyan", linewidth = 0.1)
plt.show()
```

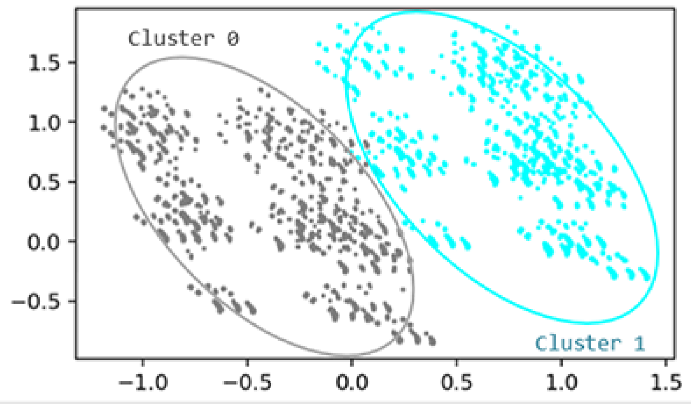


Figure 5.41 Plotting of the K-Means Clustering Result

From Figure 5.41, we can see the clusters are formed based on the distance of their data points to the centroids. While the data points in the bottom left corner of the chart are grouped to cluster 0 (grey data points), those in the upper right corner belong to cluster 1 (cyan data points). The differentiation of the two clusters are indeed quite clear based on their "locations" in this two-dimensional scatter plot. However, both the cross-tabulations and the Silhouette coefficients indicate that the characterisation of the clusters by the clustering criteria is not as straightforward as this plot suggests. We can therefore suspect that information of some input variables has gone lost during the process of dimension reduction.

Read

Refer to the link below for more details and examples on the `.transpose()` method of the pandas package:

https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.transpose.html

Refer to the link below for more details and examples on the `silhouette_score` function of the `metrics` module in the scikit-learn package:

https://scikit-learn.org/stable/modules/generated/sklearn.metrics.silhouette_score. html

Refer to the link below for more details and examples on the `PCA` estimator of the `decomposition` module in the scikit-learn package:

https://scikit-learn.org/stable/modules/generated/sklearn.decomposition.PCA.html

Chapter 3: Decision Trees

3.1 Introduction to Decision Trees

Lesson Recording

Introduction and Fitting of Decision Trees by scikit-learn

Decision trees are among the most common data mining methods which split a set (or subset) of observations to reach certain decision points based on some criteria eventually. Each decision point in the tree is also called a node and represents a subset of the sample based on which the decision tree is created. Nodes that are split from a superordinate node are called the child node while the origin node is called the parent node. A child node with no further subdivisions or splitting is called a leaf node.

Since each observation in the sample will be assigned to one of the nodes eventually, decision trees is a classification technique to separate a sample into multiple classes. The decision tree algorithm predicts the individual classification based on the values of some input variables and calculates the predicted value of the target variable at the same time. These rules of decision form the resulting model which can then be illustrated by a tree-like structure graphically. This structure convenes the interpretation of the modelling result.

We can also use the decision tree model to understand the relationship between the target variable and the input variable. In fact, decision tree handles complex relationship such as non-linearity and interaction rather well. Note that not all input variables are of the same importance in the classification process. Their hierarchy in the decision rules reflects in the decision tree in which input variables appear higher up are more important.

As mentioned, nodes without further splitting are called the leaf nodes. Their value is the prediction of the target variable for those observations classified in the corresponding nodes. If the target variable is categorical, the value of the leaf node will be the mode, the most frequent class. And it will be the mean of the data in that node if the decision tree is predicting a numeric variable.

Over the years, there have been many algorithms for splitting the nodes and hence the construction of the tree developed, proposed, and implemented in software packages. The most common ones are CHAID (chi-square automatic interaction detection), C5.0 (a proprietary algorithm) and CART (classification and regression tree). In Python, the estimator `DecisionTreeClassifier` of the scikit-learn package uses an optimised version of the CART algorithm.

As the name suggests, the CART algorithm is capable to create both classification trees and regression trees. While regression trees estimate the values of a continuous target variable, classification trees predict the outcome of a categorical target variable. In other words, CART is applicable on almost every type of output variables.

In CART, every parent node only has two child nodes. That is, CART will only split the tree into two sub-samples at every decision point. And the calculation of the split is based on the input variables that are also used to predict the target output. As the split process advances, the sample will be divided into more and more, smaller and smaller subsets. These subsets will also become more and more homogeneous. The whole splitting process will be terminated once certain stopping criteria are fulfilled.

The homogeneity of each subset reflects the split quality from a parent node to its child nodes. In classification tree, the homogeneity is measured by Gini and Entropy. Roughly speaking, both Gini and entropy measure the impurity of a node, but with different theoretical background. By comparing the impurity decrease across all possible splits in all input variables, the split with the highest reduction of impurity will be chosen. In scikit-learn, both `Gini` and `entropy` are options of the `criterion` parameter in the `DecisionTreeClassifier`.

In regression tree, the impurity is measured by the sum of squared error (SSE). The SSE is the total deviance of each observation from the sample mean. For each potential split, CART computes the SSE for each child node and the split with the lowest sum of SSE across all child nodes will be chosen. Since the mean of the target variable of the leaf node is equal to the predicted value of the target variable, splits with low SSE have child nodes that contain data whose target values are close to the mean value.

One possibility to stop the CART algorithm is when the impurity improvement of a new split drops below a certain pre-defined threshold. For instance, if a node has reached a rather low Gini or SSE respectively, meaning that the parent node itself is already rather homogeneous, another split from the node would only create homogeneous child nodes. In this case, this split is not really necessary since it does not decrease the impurity significantly.

Nevertheless, choosing the right thresholds to stop the split algorithm is not as straightforward as it seems. If the thresholds are high, the resulting tree could be oversimplified as splits become more difficult. Low thresholds, on the other hand, could lead to overcomplicated trees that are difficult to interpret and deploy.

Another possibility to stop the algorithm is when the tree has attained a pre-specified depth. The depth of the tree refers to the number of splits in it. This method can simply control the size of the tree without oversimplifying or overcomplicating it. One last stop criterion is to set a lower bound of observations in the nodes. Once the number of observations in all nodes has reached the bound, a new split from any node would only create child nodes that contain less observations than the lower bound allows. As a result, the lower bound blocks the algorithm from carrying out another split and the entire process ends.

One way to evaluate the performance of a decision tree is to examine the precision of its prediction. In other word, the predicted values of the target variable will be compared with the observed data. For classification trees, we can use the confusion matrix in which the correct and incorrect classifications are summarised. The larger the proportion of observations for which the predicted and observed classifications are identical, the more accurate is the decision tree model. For regression trees, the Root-Mean-Square-Error (RMSE) is usually used to measure the prediction accuracy of the model. Basically, the

RMSE is kind of an average deviance of all the predicted values from their observed counterparts. The lower such deviance is, the closer are the predictions to the actual values, and the better is the model.

Furthermore, the performance of a decision tree to predict unseen data must be evaluated as well. For this purpose, we partition the original dataset randomly into a training and a testing dataset. As described in Chapter 1, the training dataset is used to construct the model while the predictive performance of the model will be evaluated based on the testing dataset. In other words, the decision tree as a predictive model is evaluated by its ability to apply what it has "learned" from the training data on the testing data. If the prediction accuracy of the model on the training data is much higher than the testing data, the model tends to be overfitted. It is too specialised to the structure of the training dataset and not generalised enough for other data that do not contain certain unique characteristics of the training data.

3.2 Fitting Decision Trees

Same as K-Means clustering, scikit-learn also has an estimator for decision trees model which is integrated in the tree module. In order to be able to use all possible functions in the module, the sklearn.tree module shall be imported in the first place.

```
from sklearn import tree
```

If we want to apply a classification tree in Python, we have to initiate a `DecisionTreeClassifier` estimator object whereas if the data should be fitted by a regression tree, a `DecisionTreeRegressor` estimator object should be declared.

```
tree_Object = sklearn.tree.DecisionTreeClassifier(criterion
             = "gini", splitter = "best", max_depth = None,
             min_samples_split = 2, min_samples_leaf = 1,
             min_weight_fraction_leaf = 0.0, max_features =
             None, random_state = None, max_leaf_nodes =
             None, min_impurity_decrease = 0.0,
             min_impurity_split = None, class_weight = None)

tree_Object = sklearn.tree.DecisionTreeRegressor(criterion
             = "mse", splitter = "best", max_depth = None,
             min_samples_split = 2, min_samples_leaf = 1,
             min_weight_fraction_leaf = 0.0, max_features =
             None, random_state = None, max_leaf_nodes =
             None, min_impurity_decrease = 0.0,
             min_impurity_split = None)
```

From the above syntaxes, we can see the main differences between the estimators `DecisionTreeClassifier` and `DecisionTreeRegressor` are the values of the parameter `criterion` and the availability of the parameter `class_weight`. The following table provides description and explanation of the parameters.

Table 5.3 Parameters of the `DecisionTreeClassifier` and `DecisionTreeRegressor` Estimators

Parameter	Value Type	Description
`criterion` *(Default:* `"gini"` *for classification,* `"mse"` *for regression)*	Classification: `"gini"`, `"entropy"` Regression:`"mse"`, `"friedman_mse"`, `"mae"`, `"poisson"`	The function to measure the quality of a split. For classification trees, the supported criteria are `"gini"` for the Gini impurity and `"entropy"` for the information gain.

Parameter	Value Type	Description
		For regression trees, the supported criteria are `"mse"` for the mean squared error which is equal to variance reduction as feature selection criterion, `"friedman_mse"`, which uses mean squared error with Friedman's improvement score for potential splits, `"mae"` for the mean absolute error, and `"poisson"` which uses reduction in Poisson deviance to find splits.
`splitter` *(Default: `"best"`)*	`"best"`, `"random"`	The strategy used to choose the split at each node. Supported strategies are `"best"` to choose the best split and `"random"` to choose the best random split.
`max_depth` *(Default: `None`)*	integer	The maximum depth of the tree. If `None`, then nodes are expanded until all leaves are pure or until all leaves contain less than `min_samples_split` samples.

Parameter	Value Type	Description
`min_samples_split` *(Default: 2)*	Integer or float	The minimum number of samples required to split an internal node: If integer, consider `min_samples_split` as the minimum number. If float, `min_samples_split` is a fraction and `ceil(min_samples_split * n_samples)` are the minimum number of samples for each split.
`min_samples_leaf` *(Default: 1)*	Integer or float	The minimum number of samples required to be at a leaf node. A split point at any depth will only be considered if it leaves at least `min_samples_leaf` training samples in each of the left and right branches. This may have the effect of smoothing the model, especially in regression. If integer, consider `min_samples_leaf` as the minimum number.

Parameter	Value Type	Description
		If float, `min_samples_leaf` is a fraction and `ceil(min_samples_leaf * n_samples)` are the minimum number of samples for each node.
`min_weight_fraction_leaf` *(Default: 0.0)*	float	The minimum weighted fraction of the sum total of weights (of all the input samples) required to be at a leaf node. Samples have equal weight when `sample_weight` is not provided.
`max_features` *(Default: `None`)*	Integer, float, `"auto"`, `"sqrt"`, `"log2"`	The number of features to consider when looking for the best split: If integer, consider `max_features` features at each split. If float, `max_features` is a fraction and `int(max_features * n_features)` features are considered at each split. If `"auto"`, `max_features = sqrt(n_features)`

Parameter	Value Type	Description
		If "sqrt",max_features = sqrt(n_features) If "log2",max_features = log2(n_features). If None,max_features = n_features. Note: the search for a split does not stop until at least one valid partition of the node samples is found, even if it requires to effectively inspect more than max_features features.
random_state *(Default: None)*	integer, RandomState instance, None	Controls the randomness of the estimator. The features are always randomly permuted at each split, even if splitter is set to "best". When max_features < n_features, the algorithm will select max_features at random at each split before finding the best split among them. But the best found split may vary across different runs, even if max_features = n_features. That is the case, if the improvement of the

Parameter	Value Type	Description
		criterion is identical for several splits and one split has to be selected at random. To obtain a deterministic behaviour during fitting, `random_state` has to be fixed to an integer.
`max_leaf_nodes` *(Default: `None`)*	integer	Grow a tree with `max_leaf_nodes` in best-first fashion. Best nodes are defined as relative reduction in impurity. If `None` then unlimited number of leaf nodes.
`min_impurity_decrease` *(Default: 0.0)*	float	A node will be split if this split induces a decrease of the impurity greater than or equal to this value.
`min_impurity_split` *(Default: 0)*	float	Threshold for early stopping in tree growth. A node will split if its impurity is above the threshold, otherwise it is a leaf.
`class_weight` *(Default: `None`)* *Note: Only available for classification trees.*	dictionary, list of dictionaries, `"balanced"`	Weights associated with classes in the form `{class_label: weight}`. If `None`, all classes are supposed to have weight one. For multi-output, a list of dictionaries can be provided in

Parameter	Value Type	Description
		the same order as the columns of y. Note that for multioutput, weights should be defined for each class of every column in its own dictionary. The "balanced" mode uses the values of y to automatically adjust weights inversely proportional to class frequencies in the input data. For multi-output, the weights of each column of y will be multiplied.

(Source: https://scikit-learn.org/stable/modules/generated/sklearn.tree.DecisionTreeClassifier.html, https://scikit-learn.org/stable/modules/generated/sklearn.tree.DecisionTreeRegressor.html)

Next, we can apply the `DecisionTreeClassifier` and `DecisionTreeRegressor` estimators on the prepared training DataFrames with the input variables `X` and the target variable `y`.

```
tree_fit_Object = tree_Object.fit(X_train, y_train,
                                  sample_weight = None)
```

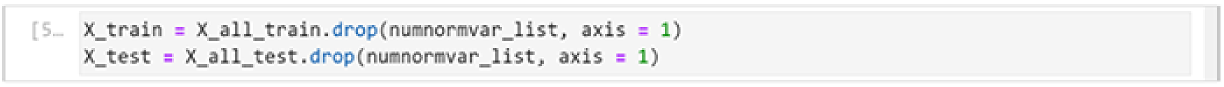
For both the `DecisionTreeClassifier` and `DecisionTreeRegressor` estimators, scikit-learn also facilitates the parameter `sample_weight` to specify individual weights in `X_train`. If it is set to `None`, the default value, all observations will be assigned equal weight. In addition, `sample_weight` will be multiplied with `class_weight` if it is specified in the `DecisionTreeClassifier` estimator for classification trees.

The fitted estimator of the decision trees algorithm is saved in `tree_fit_Object`. We can now predict the classification of the data stored in the testing DataFrame saved in `X_test`.

```
tree_pred_Object = tree_Object.predict(X_test)
```

The returned object `tree_pred_Object` is a NumPy array which contains the predicted target values of every observation in the testing dataset. The number of rows here is therefore equivalent to the number of rows in `X_test`.

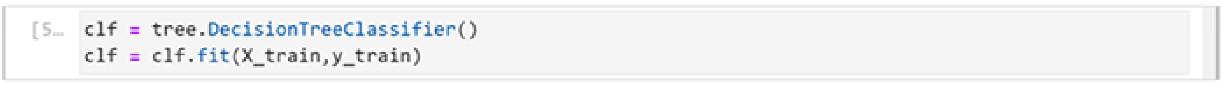
Example (Cont'd): Before constructing the decision tree, we have to remove the normalised numeric variables first since their original variables are preferred.



```
[5… X_train = X_all_train.drop(numnormvar_list, axis = 1)
    X_test = X_all_test.drop(numnormvar_list, axis = 1)
```

Figure 5.42 Create a DataFrame without the Normalised Input Variables

In the next step, we initiate an object named `clf` for the classification tree estimator `DecisionTreeClassifier` since our target variable is binary. The decision tree model will then be fitted by the `.fit()` function using the training dataset.



```
[5… clf = tree.DecisionTreeClassifier()
    clf = clf.fit(X_train,y_train)
```

Figure 5.43 Fitting Decision Trees

After the model has been created, the classification of the target variable will be predicted for the observations of both the training and testing datasets. The prediction accuracy on the training dataset is an indicator of the goodness of fit of the model while its predictive power can be assessed by its prediction accuracy on the testing dataset.

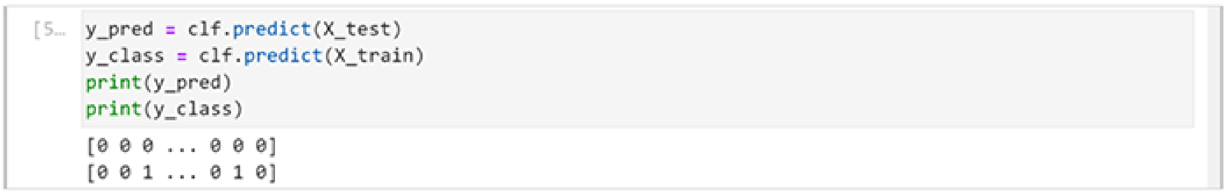

```
[5… y_pred = clf.predict(X_test)
    y_class = clf.predict(X_train)
    print(y_pred)
    print(y_class)

    [0 0 0 ... 0 0 0]
    [0 0 1 ... 0 1 0]
```

Figure 5.44 Predicting Classification of Training and Testing Datasets

The predicted classification of the testing dataset is stored in the NumPy array `y_pred` while `y_class` contains the predicted classification of the training data.

Read

Refer to the link below for more details and examples on the `DecisionTreeClassifier` estimator, the `fit()` and the `predict()` functions of the `tree` module in the scikit-learn package:

https://scikit-learn.org/stable/modules/generated/sklearn.tree.DecisionTreeClassifier.html

Refer to the link below for more details and examples on the `DecisionTreeRegressor` estimator, the `fit()` and the `predict()` functions of the `tree` module in the scikit-learn package:

https://scikit-learn.org/stable/modules/generated/sklearn.tree.DecisionTreeRegressor.html

3.3 Model Evaluation

Lesson Recording

Evaluate Decision Trees by scikit-learn

As described in Chapter 3.1, there are various possibilities to evaluate the performance of a decision tree. One of them is the confusion matrix which compares the actual with the predicted classification of the sample. In Python, the confusion matrix can be computed by the function `confusion_matrix()` from the metrics module.

```
metrics.confusion_matrix(target_var, tree_pred_Object)
```

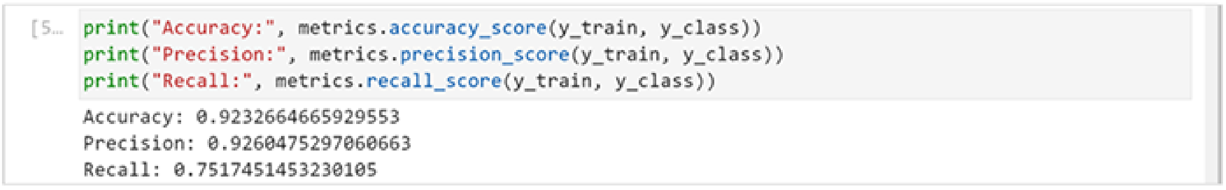
The object `target_var` is the column of the target variable in the original DataFrame and `tree_pred_Object` is the resulting NumPy array from the `predict()` function of the `DecisionTreeClassifier` or `DecisionTreeRegressor` estimator.

Other indicators such as accuracy, precision, and recall scores can also be considered for assessing the predictive performance of a decision tree.

```
metrics.accuracy_score(target_var, tree_pred_Object)
metrics.precision_score(target_var, tree_pred_Object)
metrics.recall_score(target_var, tree_pred_Object)
```

The above three measures are particularly useful in examining binary classification target variables. A binary target variable has two classes: 0 = "negative" and 1 = "positive". Accuracy score measures the sample proportion that has been classified as positive and negative correctly. Precision score, on the other hand, also called the positive predicted value (PPV) represents the sample proportion that has been predicted as positive correctly (true positive) in relation to all the cases that are predicted as positive, regardless of their actual status. Recall score, or sensitivity, is the proportion of the true positive cases in relation to the actually positive sample.

Example (Cont'd): We will first calculate the accuracy, precision and recall scores for the predicted classification of the training dataset.

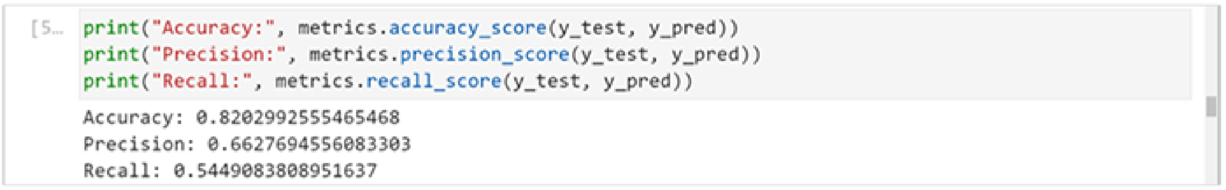


```
[5…  print("Accuracy:", metrics.accuracy_score(y_train, y_class))
     print("Precision:", metrics.precision_score(y_train, y_class))
     print("Recall:", metrics.recall_score(y_train, y_class))

     Accuracy: 0.9232664665929553
     Precision: 0.9260475297060663
     Recall: 0.7517451453230105
```

Figure 5.45 Prediction Performance on Training Data

The result seems very promising since the overall accuracy score is 0.92. Furthermore, over 92% of the observations classified in the income group with more than 50,000 USD p.a. belong really to that income class, and 75% of the individuals in the income class ">50K" are really predicted as such. As a result, the goodness of fit of the model is rather high.

Subsequently, we evaluate the predictive power of our decision tree model on the unseen data.

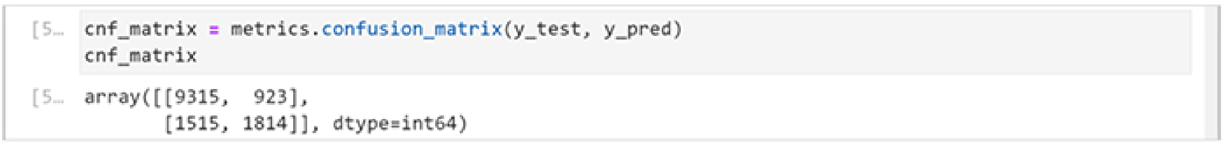


```
[5…  print("Accuracy:", metrics.accuracy_score(y_test, y_pred))
     print("Precision:", metrics.precision_score(y_test, y_pred))
     print("Recall:", metrics.recall_score(y_test, y_pred))

     Accuracy: 0.8202992555465468
     Precision: 0.6627694556083303
     Recall: 0.5449083808951637
```

Figure 5.46 Prediction Performance on Testing Data

The overall accuracy here is worse than the results of the training data prediction but remain rather high (0.82). The precision score of the prediction on the testing data is, however, on a slightly lower level (66%), while the recall score is significantly worse than the classification prediction for the training data (54%).

Another method to evaluate the predictive performance of a decision tree model is the confusion matrix.



```
[5… cnf_matrix = metrics.confusion_matrix(y_test, y_pred)
    cnf_matrix

[5… array([[9315,  923],
           [1515, 1814]], dtype=int64)
```

Figure 5.47 Confusion Matrix

Note the accuracy, precision and recall scores can all be found in the confusion matrix. The diagonal elements are the number of observations that have been classified to their own income categories accurately. This proportion is equivalent to the accuracy score calculated above: 82%. From the individuals in the ">50K" income category (3,329), only 1,814 (54%) have been classified correctly. This proportion is equal to the recall score. There are altogether 2,737 out of 13,567 observations classified in the ">50K" income category, and for 1,814 of those is the prediction accurate. The proportion of 66% is exactly the precision score.

The most important tool to understand and to evaluate a decision tree is the plot of the tree itself. The tree module of the scikit-learn package provides the `plot_tree()` function to generate such a graph conveniently.

```
tree.plot_tree(decision_tree, max_depth = None,
               feature_names = None, class_names
               = None, label = "all", filled =
               False, impurity = True, node_ids =
               False, proportion = False, rounded =
               False, precision = 3, ax = None,
               fontsize = None)
```

The following table provides description and explanation of the parameters.

Table 5.4 Parameters of the plot_tree Function

Parameter	Value Type	Description
`decision_tree` *(No default value)*	An decision tree regressor or classifier object	The decision tree to be plotted.
`max_depth` *(Default: `None`)*	integer	The maximum depth of the representation. If `None`, the tree is fully generated.
`feature_names` *(Default: `None`)*	List of strings	Names of each of the features (variables). If `None`, generic names will be used (`"X[0]", "X[1]",` …).
`class_names` *(Default: `None`)*	List of strings or Boolean	Names of each of the target classes in ascending numerical order. Only relevant for classification and not supported for multi-output. If `True`, shows a symbolic representation of the class name.
`label` *(Default: `"all"`)*	`"all"`, `"root"`, `"none"`	Whether to show informative labels for impurity, etc. Options include `"all"` to show at every node, `"root"` to show only at the top root node, or `"none"` to not show at any node.

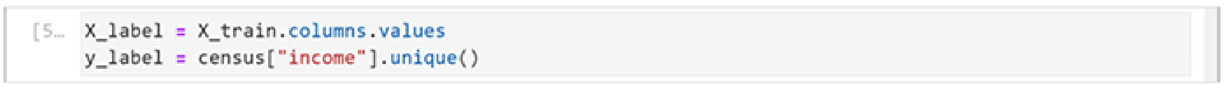
Parameter	Value Type	Description
`filled` *(Default:* `False`*)*	Boolean	When set to `True`, paint nodes to indicate majority class for classification, extremity of values for regression, or purity of node for multi-output.
`impurity` *(Default:* `True`*)*	Boolean	When set to `True`, show the impurity at each node.
`node_ids` *(Default:* `False`*)*	Boolean	When set to `True`, show the ID number on each node.
`proportion` *(Default:* `False`*)*	Boolean	When set to `True`, change the display of ”values” and/or “samples” to be proportions and percentages respectively.
`rounded` *(Default:* `False`*)*	Boolean	When set to `True`, draw node boxes with rounded corners and use Helvetica fonts instead of Times-Roman.
`precision` *(Default: 3)*	integer	Number of digits of precision for floating point in the values of impurity, threshold and value attributes of each node.

Parameter	Value Type	Description
`ax` *(Default:* `None`*)*	matplotlib axis	Axes to plot to. If `None`, use current axis. Any previous content is cleared.
`fontsize` *(Default:* `None`*)*	integer	Size of text font. If `None`, determined automatically to fit figure.

(Source: https://scikit-learn.org/stable/modules/generated/sklearn.tree.plot_tree.html)

The `plot_tree()` function can also be combined with the matplotlib options such as the plot size, the borderline settings, etc. to optimise the output of the tree.

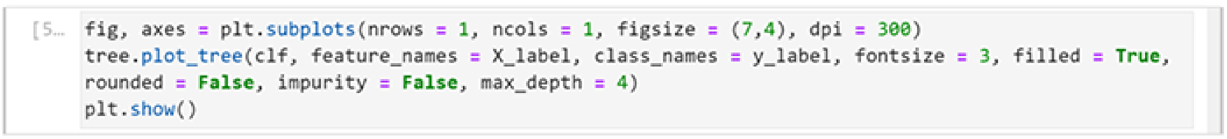
Example (Cont'd): Before plotting the decision tree, we need to prepare the labels of the input variables and the categories of the target variables. While we can use the `.columns.values` method to extract the variables names in `X_train`, we can only get the category names of the target variable by applying the `.unique()` method to extract all the unique values or strings in it.



```
[5… X_label = X_train.columns.values
    y_label = census["income"].unique()
```

Figure 5.48 Preparing Labels for Tree Plot

Finally, the decision tree can be plotted by the `tree.plot_tree()` function.



```
[5… fig, axes = plt.subplots(nrows = 1, ncols = 1, figsize = (7,4), dpi = 300)
    tree.plot_tree(clf, feature_names = X_label, class_names = y_label, fontsize = 3, filled = True,
    rounded = False, impurity = False, max_depth = 4)
    plt.show()
```

Figure 5.49 Plotting Decision Tree with `tree.plot_tree`

For the parameter `feature_names`, we can assign the array `X_label` prepared in the previous step to it since it requires the labels of all the input variables, including the dummy variables resulting from the categorical variables. The parameter `class_name` requires the labels of the categories in the target variable, which are now stored in the array `y_label`. We set the `fontsize` parameter to 3 and omit the impurity statistics (`impurity = False`) in each node so that the displayed information are still complete and readable. To visually distinguish the nodes, we instruct Python to fill the node boxes with colours (`filled = True`) while the corners do not need to be rounded (`rounded = False`). In order to keep the decision tree chart within a certain size, we limit the depth of the tree to a maximum of 4 levels (`max_depth = 4`).

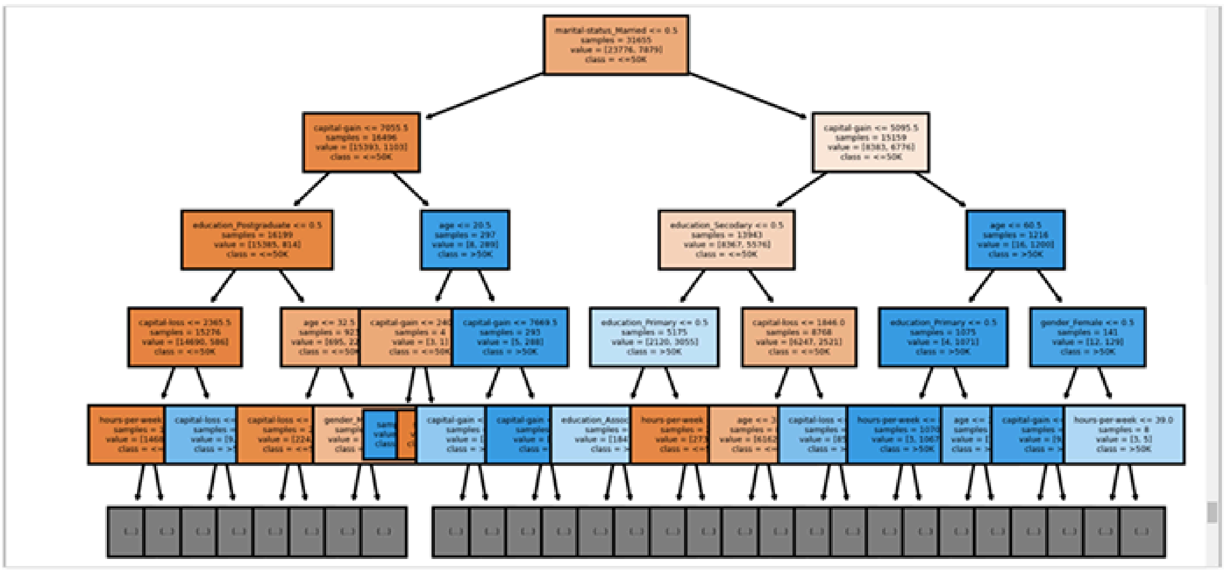


Figure 5.50 Decision Tree Plot

From Figure 5.50, we can see that the first split of the tree is generated by the marital status "Married". The "non-married" sub-sample will be assigned to the child node on the left and those "married" observations to the child node on the right. In the next step, both nodes are split by the estimated threshold values of the variable `capital-gain`. These values are different in the two child nodes. The split of the tree will then continue until certain stop criteria are fulfilled. Since we limit the tree depth down to the fourth split level in our chart, nodes of the further split levels will only be displayed in grey boxes with no proper information in it.

Read

Refer to the link below for more details and examples on the `confusion_matrix()` functions of the `metrics` module in the scikit-learn package:

https://scikit-learn.org/stable/modules/generated/sklearn.metrics.confusion_matrix.html

Refer to the link below for more details and examples on the `accuracy_score()` functions of the `metrics` module in the scikit-learn package:

https://scikit-learn.org/stable/modules/generated/sklearn.metrics.accuracy_score.html

Refer to the link below for more details and examples on the `precision_score()` functions of the `metrics` module in the scikit-learn package:

https://scikit-learn.org/stable/modules/generated/sklearn.metrics.precision_score.html

Refer to the link below for more details and examples on the `recall_score()` functions of the `metrics` module in the scikit-learn package:

https://scikit-learn.org/stable/modules/generated/sklearn.metrics.recall_score.html

Refer to the link below for more details and examples on the `plot_tree()` functions of the `tree` module in the scikit-learn package:

https://scikit-learn.org/stable/modules/generated/sklearn.tree.plot_tree.html

Summary

In this unit, we have seen how Python can be used to carry out analytics tasks based on two techniques: k-means clustering and decision trees. One of the most common packages in Python for data analytics and machine learning algorithms is scikit-learn. In scikit-learn, the analytics algorithm is called an estimator, and its parameters need to be calibrated before the fitting process are carried out. Once the calibration step is completed, we can apply the model on prepared DataFrames.

However, the available DataFrames and their contents are usually not in the format and shape that the scikit-learn algorithms require in the first place. Therefore, we need to prepare the dataset accordingly. In this unit, we have learned how to remove missing data, reduce categories, discretise numeric variables, select and rename variables, transform data, partition data into training and testing datasets and extracting dependent and independent variables from the original DataFrame.

Subsequently, the application of K-Means clustering and decision trees have been demonstrated. To understand and evaluate the classification and prediction results of the K-Means clustering, we need to generate cross-tables of the clustering criteria and the cluster indices, calculate the Silhouette coefficient, and plot the data points with optical differentiation regarding their classification on a two-dimensional scatter plot. For decision tree modules, we can create tree plots to understand the classification process and compute the confusion matrix, the scores of accuracy, precision as well as recall to assess their predictive performance.

Formative Assessment

1. What is an estimator in scikit-learn?

   a. It is the estimation algorithm of a data analytics/machine learning module.

   b. It is the function to fit a model.

   c. It is a parameter to calibrate the model estimation.

   d. It is a module of the scikit-learn package.

2. What type of tasks do most of the functions in the `metrics` module carry out?

   a. They convert imperial measurements (inch, foot, mile) to metric measurements (centimetre, metre, kilometre).

   b. They compute the mean distance of the data to the centroids in K-Means clustering.

   c. They compute metrics and distances for the evaluation of classification performance.

   d. They solve inequalities in geometry.

3. Why is partitioning data into a training and a testing dataset necessary for supervised machine learning?

   a. We can assess the predictive power of a model by applying it on unseen data.

   b. We can increase the goodness of fit of a model by creating a testing dataset.

   c. We can increase the learning ability of the algorithm.

   d. We have then two datasets to construct different models as alternatives.

4. Which function or method can be used to reduce the number of categories in a categorical variable?

   a. `.get_dummies()`

   b. `normalize()`

   c. `.deletecat()`

d. `.replace()`

5. In which of the following syntaxes will the mean and standard deviation of an object `X` be calculated during a standardisation process?
   a. `processing.standardize()`
   b. `scaler = preprocessing.StandardScaler()`
   c. `scaler.fit(X)`
   d. `scaler.transform(X)`

6. Whenever random numbers have to be drawn in scikit-learn, there is a parameter named `random_state` included in the function. What does `random_state` actually control?
   a. It draws a random number from an interval `[-1, 1]`.
   b. It creates a variable in your DataFrame to store all random numbers that have been drawn since the first run of the program.
   c. It controls the probability distribution of the random numbers.
   d. It draws the same "random numbers" in every run to make the results reproducible.

7. Which function or method can be helpful to determine the optimal number of clusters?
   a. `.transpose()`
   b. `.inertia_`
   c. `.silhouette_score()`
   d. `.PCA()`

8. How does the Principal Component Analysis reduce the dimension of an array?
   a. It projects the data in the array onto a few principal components without losing too much variation in it.
   b. It deletes those variables with a pairwise correlation larger than 0.5.

c. It removes the insignificant variables based on a regression model.

d. It selects randomly two variables and calculate the Silhouette coefficient. The pair of variables with the highest coefficients form the principal components.

9. What is not a potential stop criterion in a decision tree construction process?

   a. The tree depth

   b. The number observations in the nodes

   c. The impurity improvement

   d. The accuracy score

10. Which of the following indicators can give us some information regarding the predictive performance of a decision tree?

    a. precision score

    b. Gini and entropy

    c. Silhouette coefficient

    d. SSE and RMSE

Solutions or Suggested Answers

Formative Assessment

1. What is an estimator in scikit-learn?

   a. It is the estimation algorithm of a data analytics / machine learning module.

   Correct. The estimator in scikit-learn contains the entire algorithm of an analytics module.

   b. It is the function to fit a model.

   Incorrect. We need to apply the `.fit()` function on the estimator object to fit a model.

   c. It is a parameter to calibrate the model estimation.

   Incorrect. The estimator is the entire algorithm and not just the parameters in it.

   d. It is a module of the scikit-learn package.

   Incorrect. An estimator is part of a module of the scikit-learn package.

2. What type of tasks do most of the functions in the `metrics` module carry out?

   a. They convert imperial measurements (inch, foot, mile) to metric measurements (centimetre, metre, kilometre).

   Incorrect. There are no such functions in the `metrics` module.

   b. They compute the mean distance of the data to the centroids in K-Means clustering.

   Incorrect. The calculation of such (Euclidean) distances is integrated in the `KMeans` module.

c. They compute metrics and distances for the evaluation of classification performance.

   Correct. The functions in metrics are particularly useful in evaluating classification models.

d. They solve inequalities in geometry.

   Incorrect. There are no such functions in the `metrics` module.

3. Why is partitioning data into a training and a testing dataset necessary for supervised machine learning?

   a. We can assess the predictive power of a model by applying it on unseen data.

      Correct. Testing the prediction performance of a model by applying it on the data based on which the model was constructed is not sensible at all. Hence, we need a new dataset for this purpose.

   b. We can increase the goodness of fit of a model by creating a testing dataset.

      Incorrect. The goodness of fit of a model cannot be increased by partitioning a dataset.

   c. We can increase the learning ability of the algorithm.

      Incorrect. The learning ability cannot be enhanced by partitioning a dataset.

   d. We have then two datasets to construct different models as alternatives.

      Incorrect. Different models can also be constructed by the same dataset.

4. Which function or method can be used to reduce the number of categories in a categorical variable?

   a. `.get_dummies()`

Incorrect. The `.get_dummies()` method creates dummy variables from a categorical variable.

b. `normalize()`

Incorrect. The `.normalize()` function normalises numeric variable.

c. `.deletecat()`

Incorrect. There is no method called `.deletecat()` in the scikit-learn package.

d. `.replace()`

Correct. With the `.replace()` method, we replace the labels of various categories by one label, and the number of categories is hence reduced.

5. In which of the following syntaxes will the mean and standard deviation of an object `X` be calculated during a standardisation process?

a. `processing.standardize()`

Incorrect. There is no such syntax in the scikit-learn package at all.

b. `scaler = preprocessing.StandardScaler()`

Incorrect. This line initiates the `StandardScaler()` estimator.

c. `scaler.fit(X)`

Correct. This line computes the mean and standard deviation of the object `X`.

d. `scaler.transform(X)`

Incorrect. This line standardises the data in `X`.

6. Whenever random numbers have to be drawn in scikit-learn, there is a parameter named `random_state` included in the function. What does `random_state` actually control?

    a. It draws a random number from an interval `[-1, 1]`.

    Incorrect. The `random_state` parameter does not control the range from which the random numbers are drawn.

    b. It creates a variable in your DataFrame to store all random numbers that have been drawn since the first run of the program.

    Incorrect. There is no storage or record of the random numbers drawn in Python.

    c. It controls the probability distribution of the random numbers.

    Incorrect. The `random_state` parameter does not control the underlying distribution of the random numbers at all.

    d. It draws the same "random numbers" in every run to make the results reproducible.

    Correct. If we assign an integer to `random_state`, Python will produce the same set of "random numbers" in every run. Different integers result in different sets of "random numbers".

7. Which function or method can be helpful to determine the optimal number of clusters?

    a. `.transpose()`

    Incorrect. The `.transpose()` method swaps rows and columns of an array.

    b. `.inertia_`

Correct. Inertia measures the distortions in some clustering solutions which are used by the Elbow method. The "elbow" of the distortion plot is the optimal number of clusters.

c. `.silhouette_score()`

Incorrect. The Silhouette coefficient measures the cohesion and separation of clusters and is not used to determine the optimal number of clusters.

d. `.PCA()`

Incorrect. The PCA is a technique to reduce the dimensionality and not used to determine the optimal number of clusters.

8. How does the Principal Component Analysis reduce the dimension of an array?

a. It projects the data in the array onto a few principal components without losing too much variation in it.

Correct. The omission of the remaining, insignificant components is the core idea of dimension reduction by the PCA.

b. It deletes those variables with a pairwise correlation larger than 0.5.

Incorrect. The PCA does not delete variables directly.

c. It removes the insignificant variables based on a regression model.

Incorrect. The PCA does not require regression models for dimension reduction.

d. It selects randomly two variables and calculate the Silhouette coefficient. The pair of variables with the highest coefficients form the principal components.

Incorrect. The Silhouette coefficient measures the cohesion and separation of clusters and is not used to reduce dimensionality.

9. What is not a potential stop criterion in a decision tree construction process?

   a. The tree depth

      Incorrect. We can indeed stop the split of a decision tree if the tree has reached a certain depth.

   b. The number observations in the nodes

      Incorrect. If the sub-sample in any child node is smaller than the minimum required number of observations after a new split, the estimator will terminate the tree construction before the split.

   c. The impurity improvement

      Incorrect. The impurity improvement is actually an important stop criterion of a decision tree construction process. If new splits of a decision tree do not improve its impurity, the algorithm will stop further splitting of nodes.

   d. The accuracy score

      Correct. The accuracy score is an indicator for the predictive performance of a decision tree and not a stop criterion.

10. Which of the following indicators can give us some information regarding the predictive performance of a decision tree?

    a. precision score

       Correct. The precision score represents the sample proportion that has been predicted as positive correctly (true positive) in relation to all the cases that are predicted as positive, regardless of their actual status.

    b. Gini and entropy

       Incorrect. Gini and entropy measure the homogeneity of the nodes in a decision tree.

c. Silhouette coefficient

Incorrect. The Silhouette coefficient measures the cohesion and separation of clusters.

d. SSE and RMSE

Incorrect. SSE and RMSE are measurements of impurity for regression trees.

References

pandas. (n.d.). *pandas.crosstab*. The pandas development team. https://pandas.pydata.org/docs/reference/api/pandas.crosstab.html

pandas. (n.d.). *pandas.DataFrame.add_suffix*. The pandas development team. https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.add_suffix.html

pandas. (n.d.). *pandas.DataFrame.astype*. The pandas development team. https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.astype.html

pandas. (n.d.). *pandas.DataFrame.rename*. The pandas development team. https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.rename.html

pandas. (n.d.). *pandas.DataFrame.replace*. The pandas development team. https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.replace.html

pandas. (n.d.). *pandas.DataFrame.transpose*. The pandas development team. https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.transpose.html

pandas. (n.d.). *pandas.get_dummies*. The pandas development team. https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.get_dummies.html

scikit learn. (n.d.). *Clustering*. scikit-learn developers. https://scikit-learn.org/stable/modules/clustering.html

scikit learn. (n.d.). *Decision trees*. scikit-learn developers. https://scikit-learn.org/stable/modules/tree.html scikit learn. (n.d.). *Installing scikit-learn*. scikit-learn developers. https://scikit-learn.org/stable/install.html

scikit learn. (n.d.). *sklearn.cluster.KMeans*. scikit-learn developers. https://scikit-learn.org/stable/modules/generated/sklearn.cluster.KMeans.html

scikit learn. (n.d.). *sklearn.decomposition.PCA*. scikit-learn developers. https://scikit-learn.org/stable/modules/generated/sklearn.decomposition.PCA.html

scikit learn. (n.d.). *sklearn.metrics.accuracy_score*. scikit-learn developers. https://scikit-learn.org/stable/modules/generated/sklearn.metrics.accuracy_score.html

scikit learn. (n.d.). *sklearn.metrics.confusion_matrix*. scikit-learn developers. https://scikit-learn.org/stable/modules/generated/sklearn.metrics.confusion_matrix.html

scikit learn. (n.d.). *sklearn.metrics: Metrics*. scikit-learn developers. https://scikit-learn.org/stable/modules/classes.html?highlight=metrics#module-sklearn.metrics

scikit learn. (n.d.). *sklearn.metrics.precision_score*. scikit-learn developers. https://scikit-learn.org/stable/modules/generated/sklearn.metrics.precision_score.html

scikit learn. (n.d.). *sklearn.metrics.recall_score*. scikit-learn developers. https://scikit-learn.org/stable/modules/generated/sklearn.metrics.recall_score.html

scikit learn. (n.d.). *sklearn.metrics.silhouette_score*. scikit learn. https://scikit-learn.org/stable/modules/generated/sklearn.metrics.silhouette_score.html

scikit learn. (n.d.). *sklearn.model_selection.train_test_split*. scikit-learn developers. https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.train_test_split.html

scikit learn. (n.d.). *sklearn.preprocessing.normalize*. scikit-learn developers. https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.normalize.html

scikit learn. (n.d.). *sklearn.preprocessing: Preprocessing and normalization*. scikit-learn developers. https://scikit-learn.org/stable/modules/classes.html?highlight=preprocessing#module-sklearn.preprocessing scikit learn. (n.d.). *sklearn.preprocessing.StandardScaler*. scikit learn. https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.StandardScaler.html

scikit learn. (n.d.). *sklearn.tree.DecisionTreeClassifier*. scikit-learn developers. https://scikit-learn.org/stable/modules/generated/sklearn.tree.DecisionTreeClassifier.html

scikit learn. (n.d.). *sklearn.tree.DecisionTreeRegressor*. scikit-learn developers. https://scikit-learn.org/stable/modules/generated/sklearn.tree.DecisionTreeRegressor.html

scikit learn. (n.d.). *sklearn.tree.plot_tree*. scikit-learn developers. https://scikit-learn.org/stable/modules/generated/sklearn.tree.plot_tree.html

scikit learn. (n.d.). *User guide*. scikit-learn developers. https://scikit-learn.org/stable/user_guide.html

Study Unit 6

Basic SQL in Python

Learning Outcomes

By the end of this unit, you should be able to:

1. Explain the operations on tables in databases
2. Design Python programmes for data retrieval

Overview

In this unit, we will discuss how to use Python to execute SQL commands for database management. We will first introduce SQLite3, a standard Python package that allows programmers to use the Python environment to send SQL statements to the database. We will also discuss how we can generate SQL statements by Python programs for data query and to present their outputs as a pandas DataFrame. Eventually, we will also use SQLite3 to save the changes of the database to a physical file on the computer.

Chapter 1: Introduction to SQL and SQLite3

Lesson Recording

Introduction to SQL

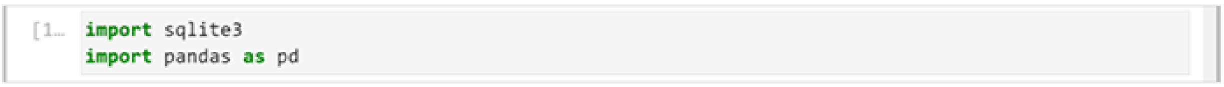
1.1 Introduction to SQL

SQL (Structured Query Language) is the most common and popular programming language designed for database management. There have been various versions of SQL including procedural extensions released such as PSQL, T-SQL, SQL/PSM, etc. Due to the large amount of internal and external data available nowadays, managing, maintaining, and updating database have become compulsory for many organisations. And the demand of SQL specialists has been increasing tremendously.

Many analytics software have integrated SQL as part of their tools. In Python, the package sqlite3 provides the possibility to embed SQL codes into Python programs to facilitate connections to databases and query the data in it. Since sqlite3 belongs to the built-in packages of Python, no installation using pip is required.

Same as scikit-learn, the analytics package introduced in the previous study unit, sqlite3 works hand-in-hand with the pandas packages since both of them are designed for data management. As a result, we can convert output table of SQL queries to pandas DataFrames and vice versa anytime.

Example (Students' score, cont'd): In this study unit, we will return to our students' examination score example from Study Units 1 & 2 and manage a database with a table containing the personal information of the students and another table in which their scores of two examinations are stored. Before we can create a database and the tables in it, we need to import the packages sqlite3 and pandas into our Python program first.



```
[1… import sqlite3
    import pandas as pd
```

Figure 6.1 Importing sqlite3 and pandas

Unlike other packages, the commands in sqlite3 must be executed through a cursor object after the connection between Python and the database has been initiated. As a result, we do not necessarily need an alias to abbreviate sqlite3 for more convenience when writing our program.

Read

Refer to the link below for more details on the sqlite3 package of Python:

https://docs.python.org/3/library/sqlite3.html

1.2 Creating Data Files in Python

Before creating a database and its tables by SQL respectively, we need to know how data entered by users at runtime of a Python program can be stored in, e.g., comma separated .csv text files. The advantage of storing data in an external file in comparison to a pandas DataFrame is that almost every software such as Excel, SPSS, SAS, etc. has a module to convert text files into its own data file format. In other words, text files are highly compatible and therefore are a good medium of data storage. Furthermore, their file size is usually comparatively small since they only contain the data and the delimiters, and no other information such as cell formatting. As a result, when exchange of data is required within an organisation or between data providers and clients, they are the most suitable format since they would not use up as much upload and download volume or time as other data file formats.

In Study Unit 1, we developed a program to let the users enter the examination scores of a course. However, the program only allows data entry but provides no mechanism to save the data in an external file. In other words, once we quit the program, all the data we have entered will not be accessible or found anymore. Here, we will introduce the Python commands to write data into a .csv file as well as to read the data from it.

In Python, we need to open an external text file with the `open()` function first before we can write data into it or read data from it.

```
with open("file_name", mode = "r/x/w/a/+") as file_object:
      instructions
```

The `with` statement is usually used in combination with the `open()` function. The data stored in the text file called `file_name` will then be stored in the `file_object` for further process. With the parameter mode, we can choose the operations that we are permitted to carry out with the file. Here is a list of some of the available modes:

Table 6.1 List of Some Available Modes of the `open()` Function

Character	Description
`"r"`	open for reading (default)
`"x"`	open for exclusive creation, failing if the file already exists

Character	Description
`"w"`	open for writing, truncating the file first
`"a"`	open for writing, appending to the end of the file if it exists
`"+"`	open for updating without truncation (reading and writing)

(Source: https://docs.python.org/3/library/functions.html#open)

Once a file is opened in a particular mode, we are only allowed to execute the permitted operations until it is reopened. For instance, if we have opened a file in reading mode, Python will block us from writing, appending, or updating any content of the file and return an error message to us if our code intends to do so. The difference between both the writing modes "w" and "a" is that "w" overwrites the entire original content in the file by our new entries while "a" appends the new entries to the end of the file while keeping the original content.

Note that it is allowed to combine the `"r"` or `"a"` modes with the `"+"` mode. In both cases, we give the permission for the file to be read and updated. The main difference here is that if we open the file in the `"r+"` mode, Python will be able to read the entire existing content from the beginning of the file and write the new entries at the end of the file. But if the file is opened in the `"a+"` mode, Python will not be able to read the entire existing content and just append the updates at the end of the file. Furthermore, if the text file does not exist, it will be created in the `"a+"` mode, while Python will return an error in the `"r+"` mode.

After the user has finished entering the data of one record, Python should write this record to the text file with the following syntax.

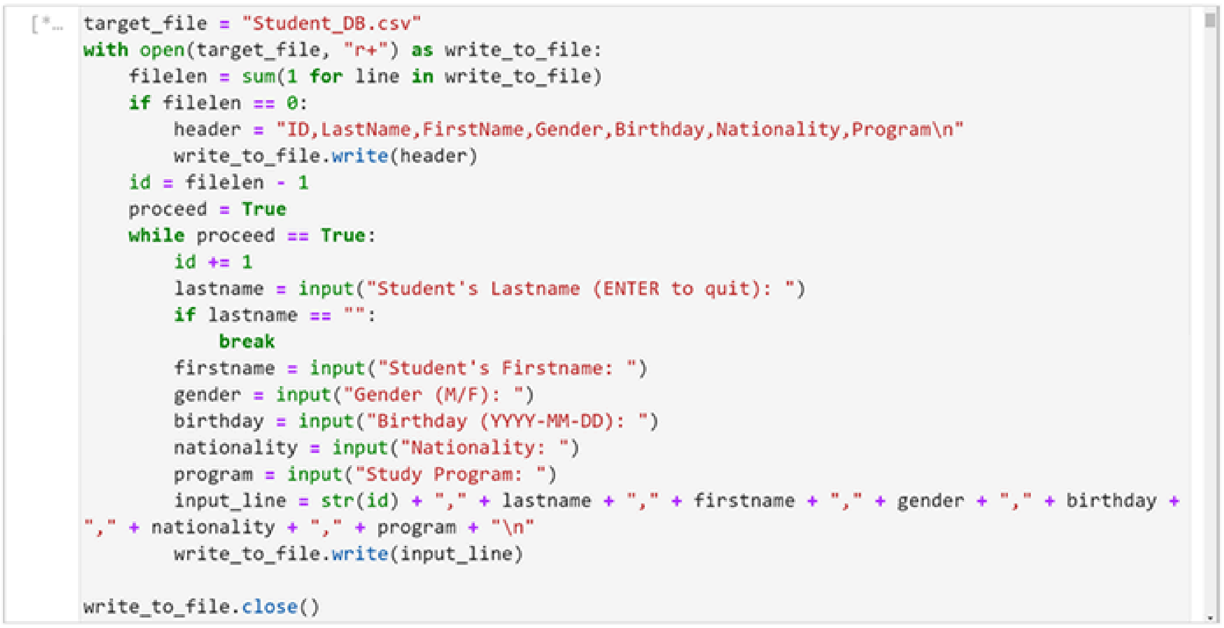
```
file_object.write(data_row)
```

The name of `file_object` must be identical with the one defined in the `with open()` statement. The object `data_row` is a string in which the entered data are stored. After all the data have been saved to the `file_object` (Python will transfer the data to the real text file in the background), we need to close the file properly to release its access to other parties.

```
file_object.close()
```

Though closing the external files may not affect the program flow directly, it is still important and a good programming habit to do so at the end of the code.

Example (Cont'd): The following program creates an interface to let the users enter the personal information of the students, including their first and last names, gender, birthday, nationality, and study programme. The input mechanism is the same as those introduced in Study Unit 1. For simplicity, we have omitted some of the control mechanisms to prevent invalid inputs here, which are actually essential and extremely important for a program to be executed correctly. After entering the data of one student, Python will convert the entries to a comma separated string and store it as a new line into the file "Student_DB.csv" for later use.

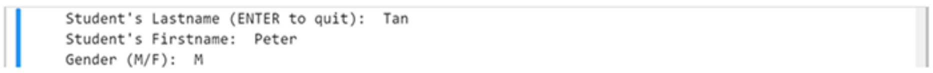


```
[*… target_file = "Student_DB.csv"
    with open(target_file, "r+") as write_to_file:
        filelen = sum(1 for line in write_to_file)
        if filelen == 0:
            header = "ID,LastName,FirstName,Gender,Birthday,Nationality,Program\n"
            write_to_file.write(header)
        id = filelen - 1
        proceed = True
        while proceed == True:
            id += 1
            lastname = input("Student's Lastname (ENTER to quit): ")
            if lastname == "":
                break
            firstname = input("Student's Firstname: ")
            gender = input("Gender (M/F): ")
            birthday = input("Birthday (YYYY-MM-DD): ")
            nationality = input("Nationality: ")
            program = input("Study Program: ")
            input_line = str(id) + "," + lastname + "," + firstname + "," + gender + "," + birthday +
    "," + nationality + "," + program + "\n"
            write_to_file.write(input_line)

    write_to_file.close()
```

Figure 6.2 Program to Input Data into a .csv File

In the first line, we store the file name in a variable called `target_file` in case we will need it multiple times in the later part of our program. By choosing the `"r+"` mode in the `with open` command in the second line, we instruct Python to open the file for reading and writing, and the existing contents of `target_file` should be stored in the object called `write_to_file`. The option of reading the stored data in the file would be particularly useful if we needed the number of existing records. This statistic is particularly helpful for two tasks: i) if the file is empty, we will need to add a line of column headers to the record which we also include in lines 4 to 6, ii) we can use the current number of records as the ID number of a new student. We would not have been able to execute these tasks if we had opened `target_file` in the `"a+"` mode instead.

Since .csv text files are comma separated, we need to insert a comma between each column when concatenating the data to a string before storing it to `target_file`. It is also important to put the escape sequence `\n` at the end of the string so that the next record will be stored in a new line. Figure 6.3 shows how the interface looks like when we run our program in JupyterLab.



```
Student's Lastname (ENTER to quit):  Tan
Student's Firstname:  Peter
Gender (M/F):  M
```

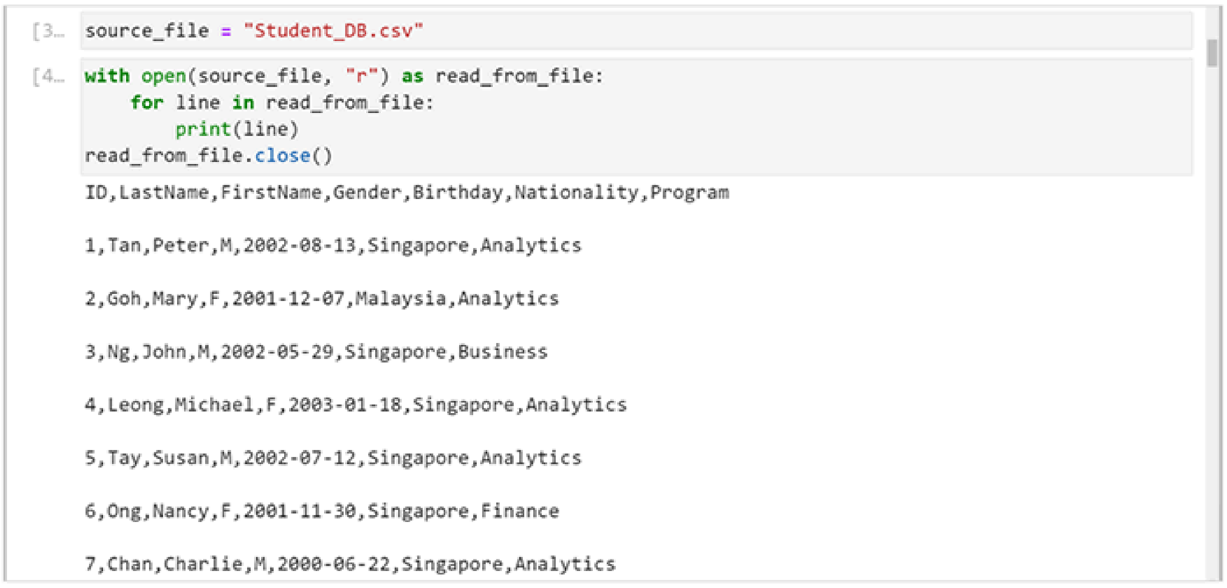
Figure 6.3 Interface to Input Data in JupyterLab

To read the existing entries of a text file, we can use a `for`-loop to go through them line by line and print the content to the screen.

```
for line in file_object:
    print(line)
```

Once again, the `file_object` must be identical with the one defined in the `with open()` statement. Note that Python is able to separate the records in the text file by recognising the line break (or escape sequence `\n`) at the end of every line. As a result, the `for`-loop automatically will run through all the lines and store the content in the variable `line`, which will then be printed to the screen by the `print()` function.

Example (Cont'd): Figure 6.4 shows our program to print the records stored in "Student_DB.csv" to the screen line by line.



```
[3… source_file = "Student_DB.csv"

[4… with open(source_file, "r") as read_from_file:
        for line in read_from_file:
            print(line)
    read_from_file.close()
```

```
ID,LastName,FirstName,Gender,Birthday,Nationality,Program

1,Tan,Peter,M,2002-08-13,Singapore,Analytics

2,Goh,Mary,F,2001-12-07,Malaysia,Analytics

3,Ng,John,M,2002-05-29,Singapore,Business

4,Leong,Michael,F,2003-01-18,Singapore,Analytics

5,Tay,Susan,M,2002-07-12,Singapore,Analytics

6,Ong,Nancy,F,2001-11-30,Singapore,Finance

7,Chan,Charlie,M,2000-06-22,Singapore,Analytics
```

Figure 6.4 Print the Records in a .csv File to the Screen Line by Line

Same as the input program, we store the file name in a variable called `source_file` in case we will still need it in the later part of our program. Subsequently, we open the file in the `"r"` mode and store the data in the object `read_from_file`. The advantage of opening the file in reading mode is that the data stored in the .csv file is safe from accidental update or removal. The data will then be printed to the screen line by line with the subsequent `for`-loop.

Read

Refer to the link below for more details and examples on the `open()` function:

https://docs.python.org/3/library/functions.html#open

Refer to the link below for more details and examples on reading and writing files in Python:

https://docs.python.org/3/tutorial/inputoutput.html#tut-files

1.3 Importing Data to SQL with SQLite3

A database is an organised collection of data, and the data are stored in tables. The data in each table is a specific set of the records in the database, such as a company's record of its employees, customers, sales, and suppliers. These tables are usually directly or indirectly connected with each other, but it is not a compulsory requirement for them to be put in the same database. Furthermore, the tables have similar structure as the pandas DataFrames: their columns represent the features of the data, or variables, and the records are stored in rows.

To work with databases in Python, we need to generate a "connection" between Python and the databases first. We can use the `connect()` function of the sqlite3 package for this purpose.

```
connection_object = sqlite3.connect("database_name")
```

If the database already exists, `connect()` will simply create a connection between the two platforms and let the user gain access to the existing database. On the other hand, if the database is new, `connect()` will create a new database with the name used in the string `"database_name"` and link it with Python directly. Once the connection has been established, we can create SQL syntaxes as strings or string variables in Python, and then send these string objects to SQL for execution. The `.cursor()` method creates a cursor object to take over this task.

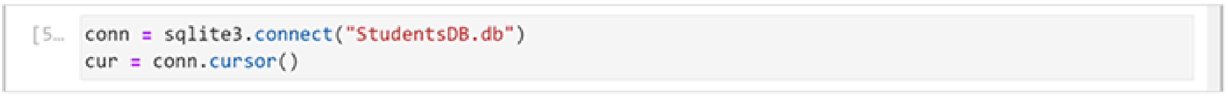
```
cursor_object = connection_object.cursor()
```

To create a table with imported data from a .csv file, we need to first read in the file as a pandas database in Python with the `pandas.read_csv()` function, which we have introduced in Chapter 1 of Study Unit 4, then send the data object to the database by the `.to_sql()` method of the pandas package.

```
data_object = pandas.read_csv("csv_file_name.csv")
data_object.to_sql("table_name", connection_object,
 if_exists)
```

With the parameter `if_exists`, we can replace (`"replace"`), append (`"append"`), or let Python create an error message (`"fail"`) if the table already exists in the database.

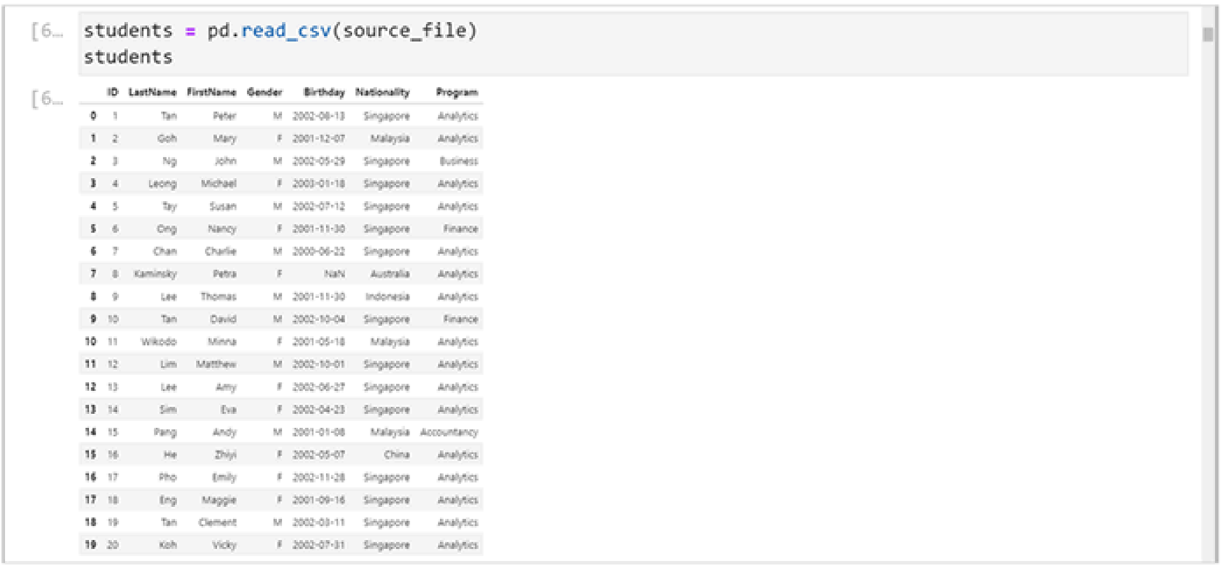
Example (Cont'd): In the first step, we establish a Python connection called `conn` to the database named `"StudentsDB.db"` and create a cursor object `cur` for later use.



```
[5…  conn = sqlite3.connect("StudentsDB.db")
     cur = conn.cursor()
```

Figure 6.5 Establish a Connection between Python and a Database

In the next step, we import the student data created with the program in Figure 6.2 as a pandas DataFrame called `students`.

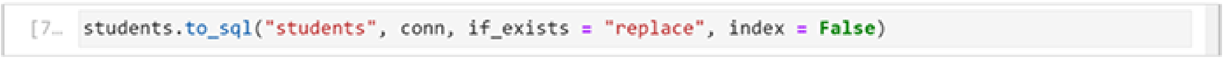


```
[6… students = pd.read_csv(source_file)
    students
```

	ID	LastName	FirstName	Gender	Birthday	Nationality	Program
0	1	Tan	Peter	M	2002-08-13	Singapore	Analytics
1	2	Goh	Mary	F	2001-12-07	Malaysia	Analytics
2	3	Ng	John	M	2002-05-29	Singapore	Business
3	4	Leong	Michael	F	2003-01-18	Singapore	Analytics
4	5	Tay	Susan	M	2002-07-12	Singapore	Analytics
5	6	Ong	Nancy	F	2001-11-30	Singapore	Finance
6	7	Chan	Charlie	M	2000-06-22	Singapore	Analytics
7	8	Kaminsky	Petra	F	NaN	Australia	Analytics
8	9	Lee	Thomas	M	2001-11-30	Indonesia	Analytics
9	10	Tan	David	M	2002-10-04	Singapore	Finance
10	11	Wikodo	Minna	F	2001-05-18	Malaysia	Analytics
11	12	Lim	Matthew	M	2002-10-01	Singapore	Analytics
12	13	Lee	Amy	F	2002-06-27	Singapore	Analytics
13	14	Sim	Eva	F	2002-04-23	Singapore	Analytics
14	15	Pang	Andy	M	2001-01-08	Malaysia	Accountancy
15	16	He	Zhiyi	F	2002-05-07	China	Analytics
16	17	Pho	Emily	F	2002-11-28	Singapore	Analytics
17	18	Eng	Maggie	F	2001-09-16	Singapore	Analytics
18	19	Tan	Clement	M	2002-03-11	Singapore	Analytics
19	20	Koh	Vicky	F	2002-07-31	Singapore	Analytics

Figure 6.6 Import a.csv file Dataset as a pandas DataFrame

Subsequently, we can create a new table in the database called `students` from this pandas DataFrame. Since we have read in the entire dataset from the .csv file, we can ask Python to replace the existing table if it exists.



```
[7… students.to_sql("students", conn, if_exists = "replace", index = False)
```

Figure 6.7 Export pandas DataFrame to Database by SQL

The parameter `index` instructs Python to write the row index as a column with column name `index_label` in the table. Since the default value here is `True`, we need to specify it in the `.to_sql()` method if we do not wish to include this column.

With the cursor object being created, we can execute the SQL commands by sending them as strings through the cursor object with the `.execute()` method.

```
cursor_object.execute("SQL_command_string")
```

Note that SQL is a separated programming language for database management and its commands are therefore not the same as those in Python. Furthermore, SQL commands are not case sensitive and should end with a semi-colon (this is usually optional, but sometimes the semi-colons are useful to separate the commands that are sent to SQL for execution at the same time).

To select a table from the database, we can send a `SELECT` statement to SQL.

```
SELECT * FROM table_name;
```

The asterisk (`*`) in the `SELECT` statement is to instruct SQL to take all columns from the table. Once the query has been carried out, we can print one record of the result to the screen by the `.fetchone()` method.

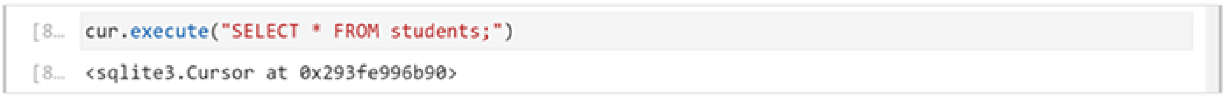
```
cursor_object.fetchone()
```

If we would like Python to print all records from the query result to the screen, we can use the `.fetchall()` method instead.

```
cursor_object.fetchall()
```

Note that once we have applied the `.fetchone()` or `.fetchall()` methods, the data records in the query result are literally fetched and no longer available. In other words, if we re-apply the `.fetchone()` or `.fetchall()` methods, we will see either no records or some of them missing. If we wish to select and check out the same table again, we will have to redo the query.

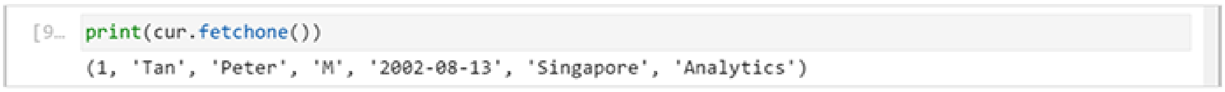
Example (Cont'd): In Figure 6.8, we use the `.execute()` method to send the `SELECT` statement to SQL for selecting the table `students` from the database.



```
[8… cur.execute("SELECT * FROM students;")
[8… <sqlite3.Cursor at 0x293fe996b90>
```

Figure 6.8 Select a Table from the Database

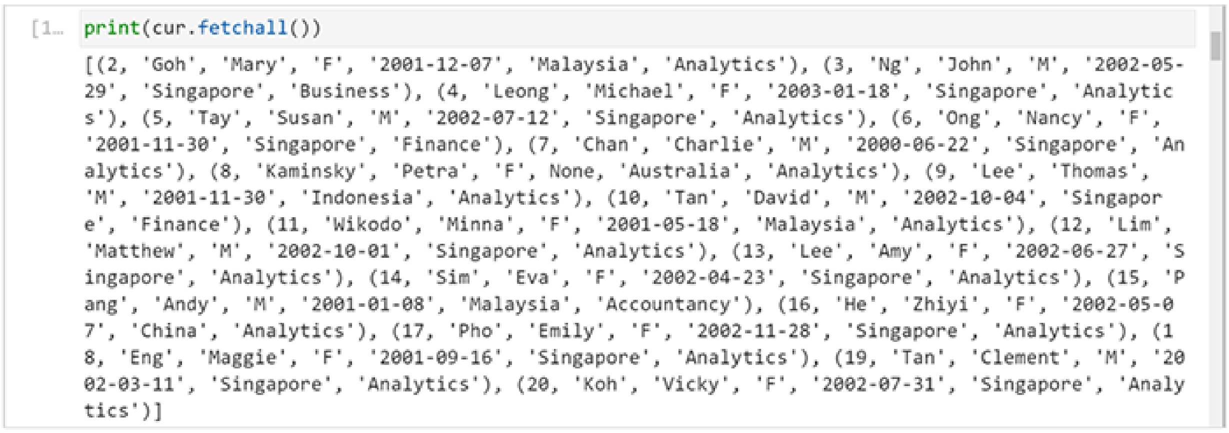
Subsequently, we print out one of the records for checking purpose.



```
[9… print(cur.fetchone())
    (1, 'Tan', 'Peter', 'M', '2002-08-13', 'Singapore', 'Analytics')
```

Figure 6.9 Fetch One Record for Printing

In the final step, we fetch all the records in the cursor object `cur` for printing.

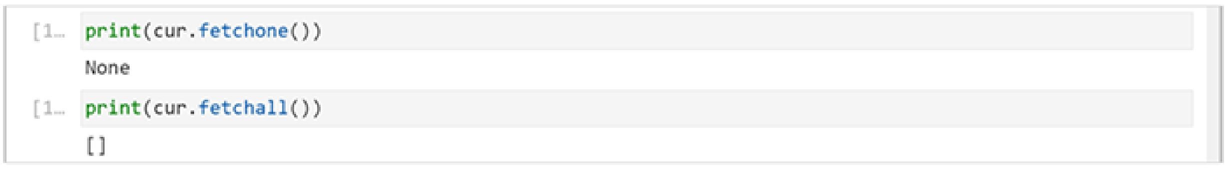


```
[1… print(cur.fetchall())
    [(2, 'Goh', 'Mary', 'F', '2001-12-07', 'Malaysia', 'Analytics'), (3, 'Ng', 'John', 'M', '2002-05-
    29', 'Singapore', 'Business'), (4, 'Leong', 'Michael', 'F', '2003-01-18', 'Singapore', 'Analytic
    s'), (5, 'Tay', 'Susan', 'M', '2002-07-12', 'Singapore', 'Analytics'), (6, 'Ong', 'Nancy', 'F',
    '2001-11-30', 'Singapore', 'Finance'), (7, 'Chan', 'Charlie', 'M', '2000-06-22', 'Singapore', 'An
    alytics'), (8, 'Kaminsky', 'Petra', 'F', None, 'Australia', 'Analytics'), (9, 'Lee', 'Thomas',
    'M', '2001-11-30', 'Indonesia', 'Analytics'), (10, 'Tan', 'David', 'M', '2002-10-04', 'Singapor
    e', 'Finance'), (11, 'Wikodo', 'Minna', 'F', '2001-05-18', 'Malaysia', 'Analytics'), (12, 'Lim',
    'Matthew', 'M', '2002-10-01', 'Singapore', 'Analytics'), (13, 'Lee', 'Amy', 'F', '2002-06-27', 'S
    ingapore', 'Analytics'), (14, 'Sim', 'Eva', 'F', '2002-04-23', 'Singapore', 'Analytics'), (15, 'P
    ang', 'Andy', 'M', '2001-01-08', 'Malaysia', 'Accountancy'), (16, 'He', 'Zhiyi', 'F', '2002-05-0
    7', 'China', 'Analytics'), (17, 'Pho', 'Emily', 'F', '2002-11-28', 'Singapore', 'Analytics'), (1
    8, 'Eng', 'Maggie', 'F', '2001-09-16', 'Singapore', 'Analytics'), (19, 'Tan', 'Clement', 'M', '20
    02-03-11', 'Singapore', 'Analytics'), (20, 'Koh', 'Vicky', 'F', '2002-07-31', 'Singapore', 'Analy
    tics')]
```

Figure 6.10 Fetch All Records for Printing

From Figure 6.10, we can see that the first record has already been fetched in Figure 6.9 and is therefore not included in the output.

If we re-fetch the records from the query output after applying `fetchall()` once before, SQL will return an empty object to us.



```
[1… print(cur.fetchone())
    None
[1… print(cur.fetchall())
    []
```

Figure 6.11 Re-Fetch Records after Applying `fetchone()` or `fetchall()`

Read

Refer to the link below for more details and examples on the `to_sql()` function of the pandas package:

https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.to_sql.html

Refer to the link below for more details and examples on the `connect()`, `.cursor()`, `.execute()`, `.fetchone()`, and `.fetchall()` functions and methods of the sqlite3 package:

https://docs.python.org/3/library/sqlite3.html

Chapter 2: Data Query

Lesson Recording

Data Query with SQL

2.1 Selecting Table

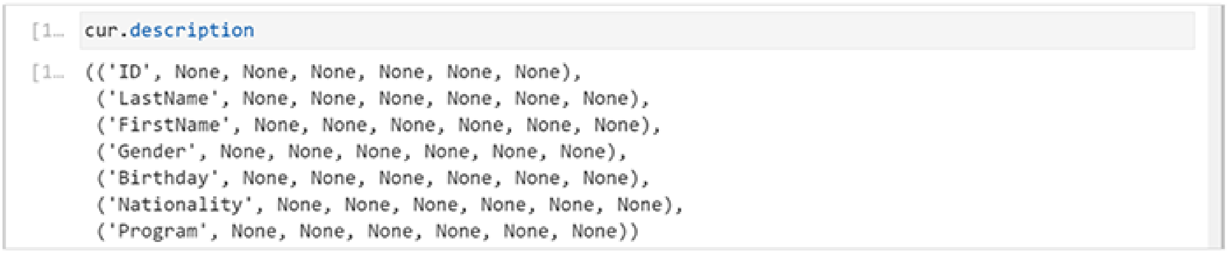
In Chapter 1, we have already introduced the `SELECT` statement of the standard SQL in its simplest form for table selection. In the following, we will discuss some further options provided by the `SELECT` statement to optimise our data query.

The `SELECT` statement also allows us to select some of the variables from the table instead of all of them. But in some cases, we may not even know the variables that the table contains or how their names are correctly spelt. In this case, we can use the `.description` attribute to extract the variable names from the last queried table.

```
cursor_object.description
```

Note that `.description` is an attribute and not a method. As a result, there are no brackets and arguments behind it. The returned object is a collection of tuples where the first item of each tuple is the column name, and the last six items are `None`.

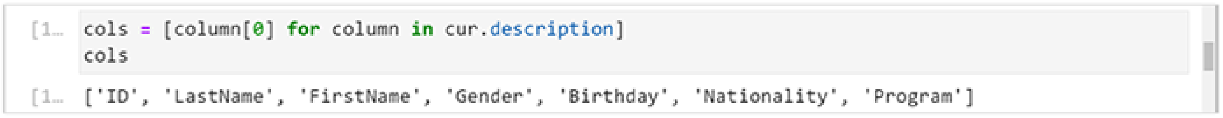
Example (Cont'd): In Figure 6.12, we will use the `.description` attribute to extract the column names of the table `students`.



```
[1… cur.description

[1… (('ID', None, None, None, None, None, None),
     ('LastName', None, None, None, None, None, None),
     ('FirstName', None, None, None, None, None, None),
     ('Gender', None, None, None, None, None, None),
     ('Birthday', None, None, None, None, None, None),
     ('Nationality', None, None, None, None, None, None),
     ('Program', None, None, None, None, None, None))
```

Figure 6.12 Extract Column Names of a Table by the `.description` Attribute

Unfortunately, the returned object does not have a particularly useful form for further usage. Since we only need the first item of each tuple, we can run a for-loop within a list to extract it.



```
[1… cols = [column[0] for column in cur.description]
    cols

[1… ['ID', 'LastName', 'FirstName', 'Gender', 'Birthday', 'Nationality', 'Program']
```

Figure 6.13 Generate Variable List of a Table

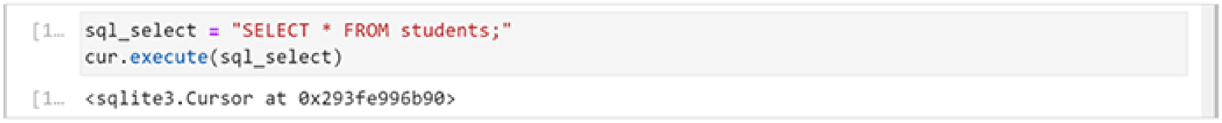
While the `for`-loop is running through the `cur.description` object, the current tuple is stored in the object `column`, from which the item with the index 0 will then be put in the list named `cols`.

From the previous Chapter, we learn that we can fetch the records from a table for further processes such as printing. However, the data are stored in tuples and when they are printed, we do not see them as table such as a pandas DataFrame. Furthermore, once they are fetched, our program has no more access to the queried table. As a result, it may be desirable to store the result of the query in a pandas DataFrame. In fact, pandas provides the method `.from_records()` for this purpose.

```
query_object = pd.DataFrame.from_records
              (data = cursor_object.fetchall(), columns)
```

The `.from_records()` method is actually created to convert structured or n-dimensional record arrays to pandas DataFrames. Nevertheless, it suits our purpose perfectly by passing the resulting object of the `fetchall()` function as the parameter `data` to the `.from_records()` method. With the `columns` parameter, we can specify our own column names of the output DataFrame. The default value here is `None`, and the corresponding column names would be simply the column indices.

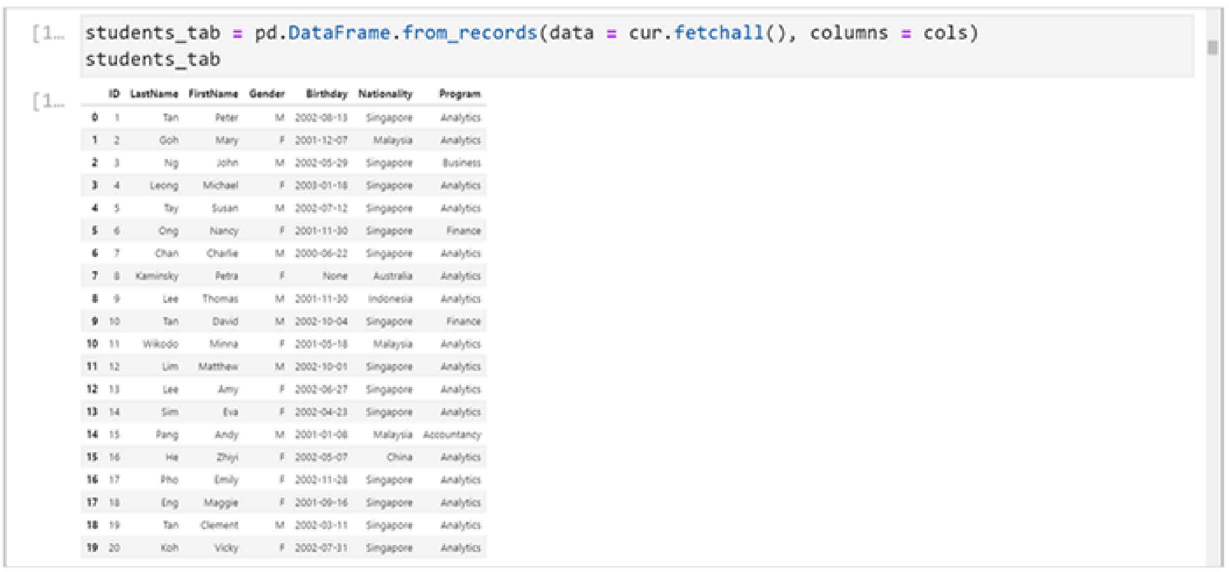
Example (Cont'd): We can now start a new data query by selecting the entire table `students` and then store the result of the query in a pandas DataFrame. The query step is almost identical to the syntax in Figure 6.8. The only difference here is that the `.execute()` method will not carry out our SQL command directly, but a string variable called `sql_select` in which the `SELECT` statement is stored for future use.



```
[1… sql_select = "SELECT * FROM students;"
    cur.execute(sql_select)

[1… <sqlite3.Cursor at 0x293fe996b90>
```

Figure 6.14 Execute an SQL Statement Stored in a String Variable

In the next step, we can generate the data by applying the `.fetchall()` method on the cursor object and convert it to a pandas DataFrame. We will also specify the list `cols` which was generated in Figure 6.13 as our column names here.



```
[1… students_tab = pd.DataFrame.from_records(data = cur.fetchall(), columns = cols)
    students_tab
```

[1…

	ID	LastName	FirstName	Gender	Birthday	Nationality	Program
0	1	Tan	Peter	M	2002-08-13	Singapore	Analytics
1	2	Goh	Mary	F	2001-12-07	Malaysia	Analytics
2	3	Ng	John	M	2002-05-29	Singapore	Business
3	4	Leong	Michael	F	2003-01-18	Singapore	Analytics
4	5	Tay	Susan	M	2002-07-12	Singapore	Analytics
5	6	Ong	Nancy	F	2001-11-30	Singapore	Finance
6	7	Chan	Charlie	M	2000-06-22	Singapore	Analytics
7	8	Kaminsky	Petra	F	None	Australia	Analytics
8	9	Lee	Thomas	M	2001-11-30	Indonesia	Analytics
9	10	Tan	David	M	2002-10-04	Singapore	Finance
10	11	Wikodo	Minna	F	2001-05-18	Malaysia	Analytics
11	12	Lim	Matthew	M	2002-10-01	Singapore	Analytics
12	13	Lee	Amy	F	2002-06-27	Singapore	Analytics
13	14	Sim	Eva	F	2002-04-23	Singapore	Analytics
14	15	Pang	Andy	M	2001-01-08	Malaysia	Accountancy
15	16	He	Zhiyi	F	2002-05-07	China	Analytics
16	17	Pho	Emily	F	2002-11-28	Singapore	Analytics
17	18	Eng	Maggie	F	2001-09-16	Singapore	Analytics
18	19	Tan	Clement	M	2002-03-11	Singapore	Analytics
19	20	Koh	Vicky	F	2002-07-31	Singapore	Analytics

Figure 6.15 Convert the Result of an SQL Query to a pandas DataFrame

Figure 6.15 shows the table `students` after being converted to a pandas DataFrame, a format that has become well-known to us from the previous study units.

Read

Refer to the link below for more details and examples on the `.description` attribute of the sqlite3 package:

https://docs.python.org/3/library/sqlite3.html#sqlite3.Cursor.description

Refer to the link below for more details and examples on the `.from_records()` function of the pandas package:

https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.from_records.html

Refer to the links below for more details and examples on the `SELECT` statement of SQL:

https://www.w3schools.com/sql/sql_select.asp https://www.sqlitetutorial.net/sqlite-select/

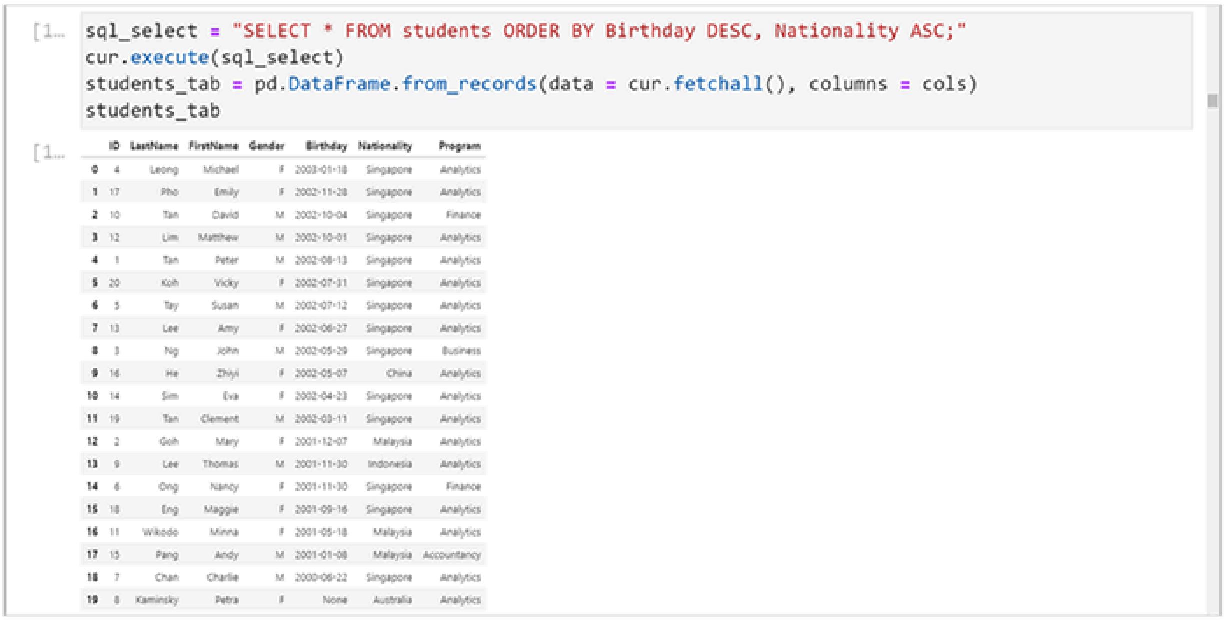
2.2 Sorting Data

In SQL, we can add the keyword `ORDER BY` to the `SELECT` statement to sort the data of a table by some of its variables in the ascending or descending orders.

```
SELECT * FROM table_name
ORDER BY var1_name, var2_name ASC|DESC;
```

Note that if we intend to sort the table by multiple variables, we will need to separate their names by commas. The sequence of the variables in this list also reflects the sorting hierarchy. That is, the data are sorted by the first variable in the list initially, those tied records are then sorted by the second variable, and so on. If the data have to be sorted in the descending order by a particular variable, we must specify the `DESC` option behind the variable name. Since the default value here is `ASC`, we can omit this option for a variable if the data should be sorted in its ascending order.

Example (Cont'd): Suppose we would like to sort the students by their age and nationality in the table `students`. Since the table only contains their birthdays, stored in the variable `Birthday`, we need to sort the records in the descending order so that their age will be ordered naturally.



```
[1… sql_select = "SELECT * FROM students ORDER BY Birthday DESC, Nationality ASC;"
     cur.execute(sql_select)
     students_tab = pd.DataFrame.from_records(data = cur.fetchall(), columns = cols)
     students_tab
[1…
```

	ID	LastName	FirstName	Gender	Birthday	Nationality	Program
0	4	Leong	Michael	F	2003-01-18	Singapore	Analytics
1	17	Pho	Emily	F	2002-11-28	Singapore	Analytics
2	10	Tan	David	M	2002-10-04	Singapore	Finance
3	12	Lim	Matthew	M	2002-10-01	Singapore	Analytics
4	1	Tan	Peter	M	2002-08-13	Singapore	Analytics
5	20	Koh	Vicky	F	2002-07-31	Singapore	Analytics
6	5	Tay	Susan	M	2002-07-12	Singapore	Analytics
7	13	Lee	Amy	F	2002-06-27	Singapore	Analytics
8	3	Ng	John	M	2002-05-29	Singapore	Business
9	16	He	Zhiyi	F	2002-05-07	China	Analytics
10	14	Sim	Eva	F	2002-04-23	Singapore	Analytics
11	19	Tan	Clement	M	2002-03-11	Singapore	Analytics
12	2	Goh	Mary	F	2001-12-07	Malaysia	Analytics
13	9	Lee	Thomas	M	2001-11-30	Indonesia	Analytics
14	6	Ong	Nancy	F	2001-11-30	Singapore	Finance
15	18	Eng	Maggie	F	2001-09-16	Singapore	Analytics
16	11	Wikodo	Minna	F	2001-05-18	Malaysia	Analytics
17	15	Pang	Andy	M	2001-01-08	Malaysia	Accountancy
18	7	Chan	Charlie	M	2000-06-22	Singapore	Analytics
19	8	Kaminsky	Petra	F	None	Australia	Analytics

Figure 6.16 Sort Students by Their Birthdays and Nationalities

The second sorting criterion, `Nationality`, is put after a comma to separate it from the other sorting criteria. The option `ASC` can actually be omitted here. From the result, we can see that the students with ID number 9 and ID number 6 share the same birthday. Since student number 9 is from Indonesia, which is alphabetically before Singapore, this record is sorted in front of the student number 6. It is also noteworthy that the birthday of the student number 8 is missing. As a result, it appears at the end of the table after sorting the students in the descending order of their birthdays.

Read

Refer to the links below for more details and examples on the `ORDER BY` option of SQL:

https://www.w3schools.com/sql/sql_orderby.asp

https://www.sqlitetutorial.net/sqlite-order-by/

2.3 Filtering Data

The main purpose of data query is to request the records of interest from the available tables. And more often, it is not the entire table that we are actually looking for, instead we would like to have records that fulfil certain criteria. In SQL, we can use the `WHERE` clause in the `SELECT` statement to filter the useful records for us.

```
SELECT * FROM table_name
WHERE var_name = value;
```

In the above syntax, the selection criterion is presented in its simplest form: records will only be selected if one of the variables is equal to a certain value. We can also construct other criteria by using the operators listed in the following table.

Table 6.2 Operators in the `WHERE` Clause

Operator	Description
=	Equal
>	Greater than

Operator	**Description**
`<`	Less than
`>=`	Greater than or equal
`<=`	Less than or equal
`<>`	Not equal (Note: In some SQL versions it may be written as `!=`)
`BETWEEN`	Between a certain range
`LIKE`	Search for a pattern
`IN`	To specify multiple possible values for a column

(Source: https://www.w3schools.com/sql/sql_where.asp)

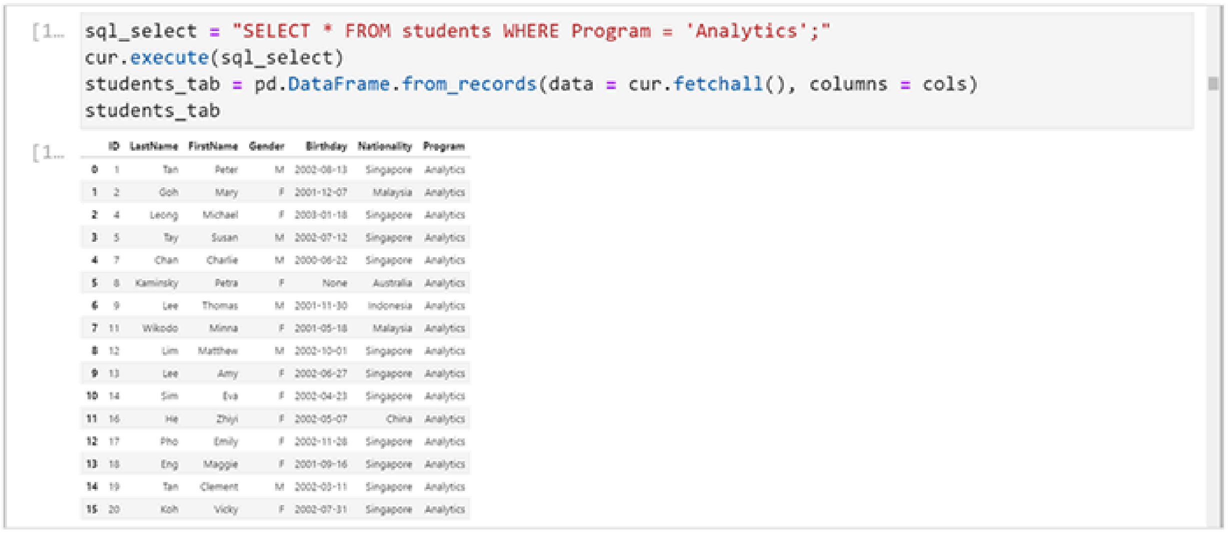
Same as the `if`-command in Python, we can also link multiple criteria in one statement using the `AND`, `OR` and `NOT` operators.

Sometimes, we would rather not obtain records that contain missing values in one or more variables from a query. In this case, we need to add the `IS NOT NULL` syntax to the `WHERE` clause.

```
SELECT * FROM table_name
WHERE var_name IS NOT NULL;
```

If the statement were written without the `NOT` operator, SQL would return all records with missing values in the variable `var_name` to us.

Example (Cont'd): Suppose we would like to select students of the analytics programme from the table `students`. The query can simply be carried out by adding the criterion `Program = 'Analytics'` to the `WHERE` syntax.



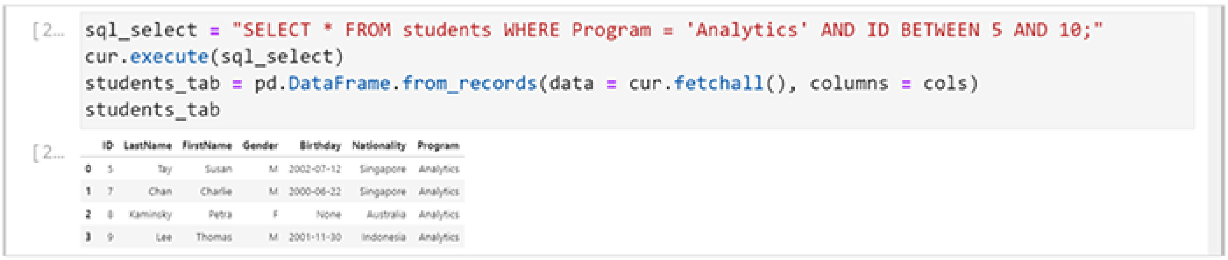
```
sql_select = "SELECT * FROM students WHERE Program = 'Analytics';"
cur.execute(sql_select)
students_tab = pd.DataFrame.from_records(data = cur.fetchall(), columns = cols)
students_tab
```

```
    ID  LastName  FirstName  Gender    Birthday  Nationality    Program
0    1       Tan      Peter       M  2002-08-13    Singapore  Analytics
1    2       Goh       Mary       F  2001-12-07     Malaysia  Analytics
2    4     Leong    Michael       F  2003-01-18    Singapore  Analytics
3    5       Tay      Susan       M  2002-07-12    Singapore  Analytics
4    7      Chan    Charlie       M  2000-06-22    Singapore  Analytics
5    8  Kaminsky      Petra       F        None    Australia  Analytics
6    9       Lee     Thomas       M  2001-11-30    Indonesia  Analytics
7   11    Wikodo      Minna       F  2001-05-18     Malaysia  Analytics
8   12       Lim    Matthew       M  2002-10-01    Singapore  Analytics
9   13       Lee        Amy       F  2002-06-27    Singapore  Analytics
10  14       Sim        Eva       F  2002-04-23    Singapore  Analytics
11  16        He      Zhiyi       F  2002-05-07        China  Analytics
12  17       Pho      Emily       F  2002-11-28    Singapore  Analytics
13  18       Eng     Maggie       F  2001-09-16    Singapore  Analytics
14  19       Tan    Clement       M  2002-03-11    Singapore  Analytics
15  20       Koh      Vicky       F  2002-07-31    Singapore  Analytics
```

Figure 6.17 Select only Analytics Students from the Table `students`

If the value is a string such as "Analytics" in the above syntax, we will need to put it in a pair of quotation marks. And it is important here to pay attention to when and where single or double quotation marks should be used.

Suppose we would like to narrow down our query to only analytics students with ID numbers between 5 and 10.

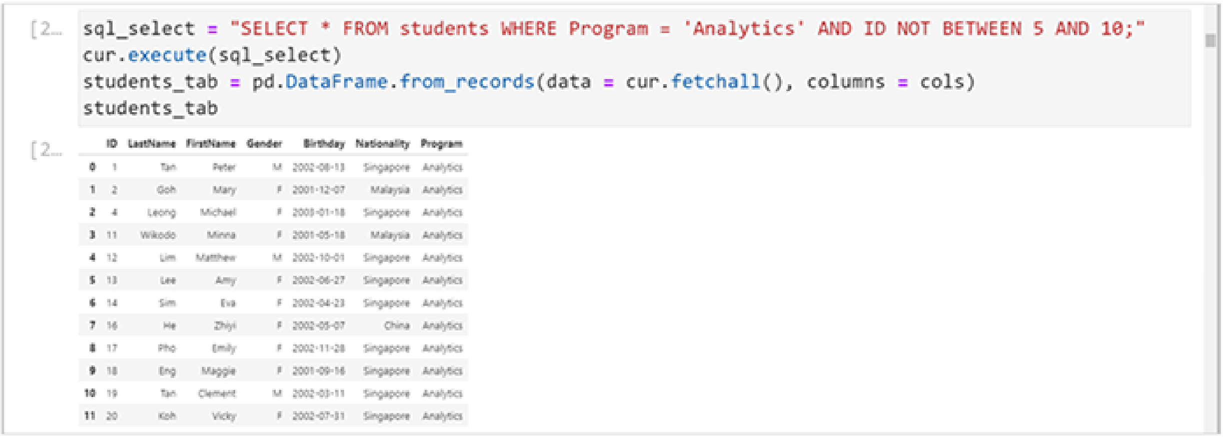


```
sql_select = "SELECT * FROM students WHERE Program = 'Analytics' AND ID BETWEEN 5 AND 10;"
cur.execute(sql_select)
students_tab = pd.DataFrame.from_records(data = cur.fetchall(), columns = cols)
students_tab
```

```
   ID  LastName  FirstName  Gender    Birthday  Nationality    Program
0   5       Tay      Susan       M  2002-07-12    Singapore  Analytics
1   7      Chan    Charlie       M  2000-06-22    Singapore  Analytics
2   8  Kaminsky      Petra       F        None    Australia  Analytics
3   9       Lee     Thomas       M  2001-11-30    Indonesia  Analytics
```

Figure 6.18 Select only Analytics Students with ID number between 5 And 10

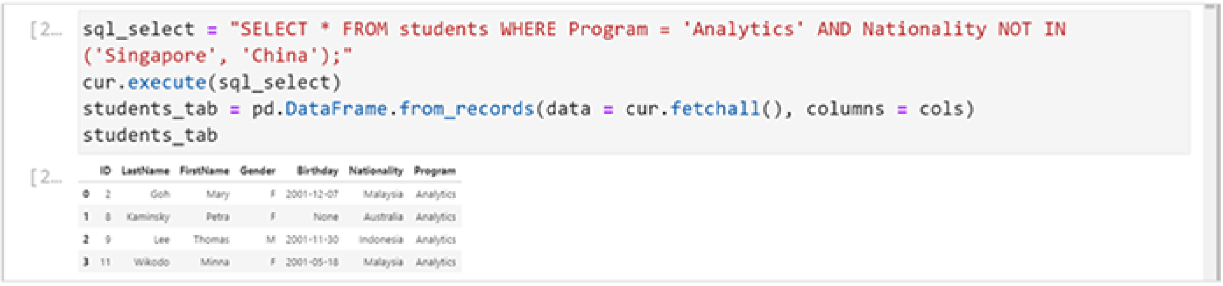
Suppose these are not the students that we are actually looking for, and we would like to select the other analytics students. All we need to modify in the above syntax is to change the operator from `BETWEEN` to `NOT BETWEEN` for the `ID` variable.



```
[2…  sql_select = "SELECT * FROM students WHERE Program = 'Analytics' AND ID NOT BETWEEN 5 AND 10;"
     cur.execute(sql_select)
     students_tab = pd.DataFrame.from_records(data = cur.fetchall(), columns = cols)
     students_tab

[2…      ID  LastName  FirstName  Gender    Birthday  Nationality    Program
      0   1       Tan      Peter       M  2002-08-13    Singapore  Analytics
      1   2       Goh       Mary       F  2001-12-07     Malaysia  Analytics
      2   4     Leong    Michael       F  2003-01-18    Singapore  Analytics
      3  11    Wikodo      Minna       F  2001-05-18     Malaysia  Analytics
      4  12       Lim    Matthew       M  2002-10-01    Singapore  Analytics
      5  13       Lee        Amy       F  2002-06-27    Singapore  Analytics
      6  14       Sim        Eva       F  2002-04-23    Singapore  Analytics
      7  16        He      Zhiyi       F  2002-05-07        China  Analytics
      8  17       Pho      Emily       F  2002-11-28    Singapore  Analytics
      9  18       Eng     Maggie       F  2001-09-16    Singapore  Analytics
     10  19       Tan    Clement       M  2002-03-11    Singapore  Analytics
     11  20       Koh      Vicky       F  2002-07-31    Singapore  Analytics
```

Figure 6.19 Select only Analytics Students with ID number not between 5 And 10

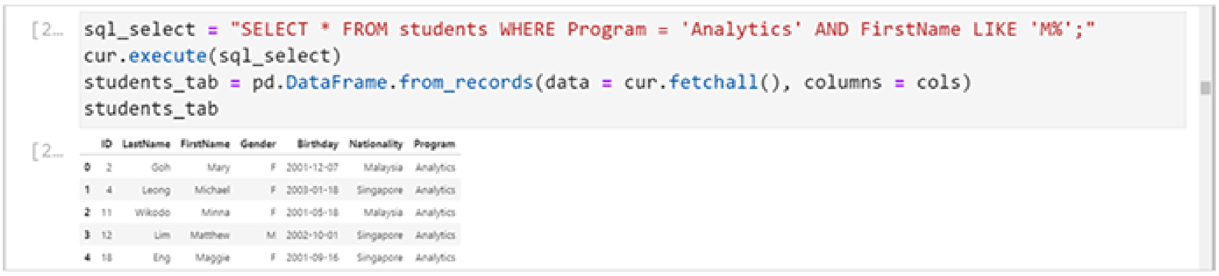
In the next query, we would like to select all analytics students who are not from Singapore or China. With the `IN` operator, we can specify the two values "Singapore" and "China" that we are searching for in the column `Nationality`. Note that these values must be put in a pair of round brackets. Finally, the `NOT` operator should be added to the syntax `IN ('Singapore', 'China')` to negate it.



```
[2…  sql_select = "SELECT * FROM students WHERE Program = 'Analytics' AND Nationality NOT IN
     ('Singapore', 'China');"
     cur.execute(sql_select)
     students_tab = pd.DataFrame.from_records(data = cur.fetchall(), columns = cols)
     students_tab

[2…     ID  LastName  FirstName  Gender    Birthday  Nationality    Program
      0  2       Goh       Mary       F  2001-12-07     Malaysia  Analytics
      1  8  Kaminsky      Petra       F        None    Australia  Analytics
      2  9       Lee     Thomas       M  2001-11-30    Indonesia  Analytics
      3 11    Wikodo      Minna       F  2001-05-18     Malaysia  Analytics
```

Figure 6.20 Select Analytics Students not from Singapore or China

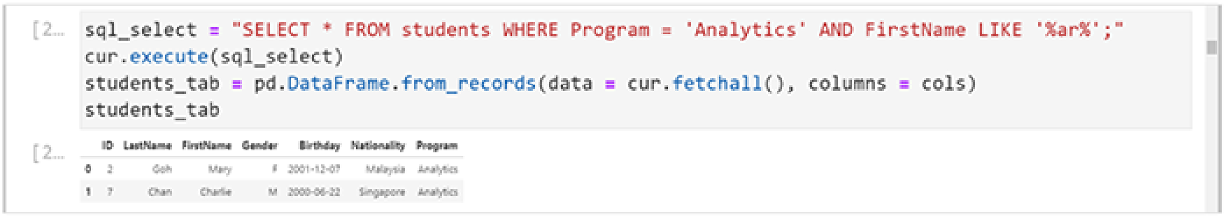
Now, all analytics students whose first names start with an "M" should be selected. Here, we can use the `LIKE` operator and the value is `'M%'`. The (%) sign is a wildcard that represents zero, one, or multiple characters. In other words, our value is an `M` followed by a string of arbitrary length.



```
[2… sql_select = "SELECT * FROM students WHERE Program = 'Analytics' AND FirstName LIKE 'M%';"
    cur.execute(sql_select)
    students_tab = pd.DataFrame.from_records(data = cur.fetchall(), columns = cols)
    students_tab

[2…    ID LastName FirstName Gender    Birthday Nationality   Program
    0   2      Goh      Mary      F  2001-12-07    Malaysia Analytics
    1   4    Leong   Michael      F  2003-01-18   Singapore Analytics
    2  11   Wikodo     Minna      F  2001-05-18    Malaysia Analytics
    3  12      Lim   Matthew      M  2002-10-01   Singapore Analytics
    4  18      Eng    Maggie      F  2001-09-16   Singapore Analytics
```

Figure 6.21 Select Analytics Students with First Names that Start with "M"

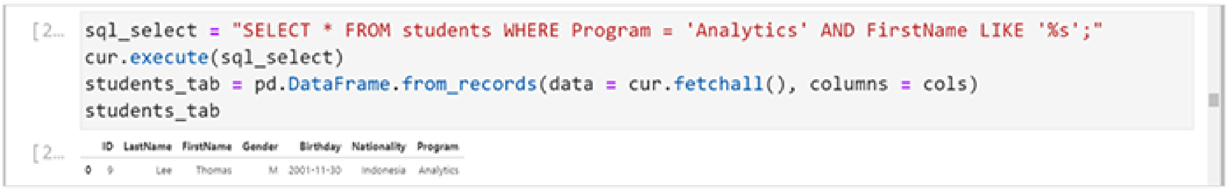
Instead of searching for first names starting with a particular letter, we can also select records with first names that contain a pre-defined string, which is "ar" in the following example.



```
[2… sql_select = "SELECT * FROM students WHERE Program = 'Analytics' AND FirstName LIKE '%ar%';"
    cur.execute(sql_select)
    students_tab = pd.DataFrame.from_records(data = cur.fetchall(), columns = cols)
    students_tab

[2…    ID LastName FirstName Gender    Birthday Nationality   Program
    0   2      Goh      Mary      F  2001-12-07    Malaysia Analytics
    1   7     Chan   Charlie      M  2000-06-22   Singapore Analytics
```

Figure 6.22 Select Analytics Students with First Names that Contain "ar"

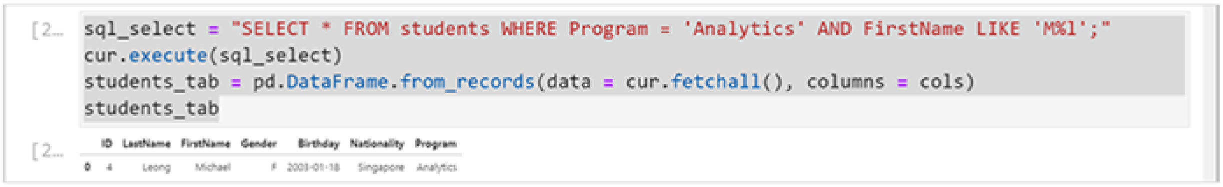
Same as our code in Figure 6.22, we can use the (%) wildcard within the string to specify the position of our pre-defined string. Unsurprisingly, we can also select first names that end with a certain character, which is "s" here.



```
[2… sql_select = "SELECT * FROM students WHERE Program = 'Analytics' AND FirstName LIKE '%s';"
    cur.execute(sql_select)
    students_tab = pd.DataFrame.from_records(data = cur.fetchall(), columns = cols)
    students_tab

[2…    ID LastName FirstName Gender    Birthday Nationality   Program
    0   9      Lee    Thomas      M  2001-11-30   Indonesia Analytics
```

Figure 6.23 Select Analytics Students with First Names that End with "s"

If we wish to select first names that start and end with some pre-defined characters, we can put the wildcard (%) between the start and the end characters. In the following example, we would like to select students with first names that start with an "M" and end with an "l".

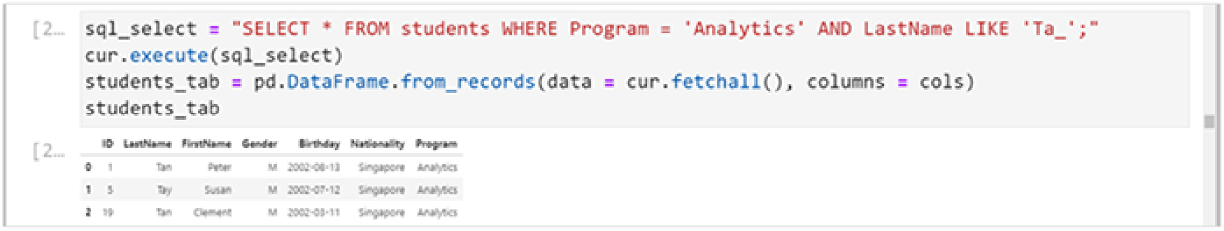


```
[2… sql_select = "SELECT * FROM students WHERE Program = 'Analytics' AND FirstName LIKE 'M%l';"
    cur.execute(sql_select)
    students_tab = pd.DataFrame.from_records(data = cur.fetchall(), columns = cols)
    students_tab
```

```
[2…    ID  LastName  FirstName  Gender     Birthday  Nationality   Program
    0   4     Leong    Michael       F   2003-01-18    Singapore  Analytics
```

Figure 6.24 Select Analytics Students with First Names that Start with "M" and End with "l"

The other type of wildcard for the `LIKE` operator is the underscore sign (_), which represents a single character. In the following example, we would like to select all students whose last names start with "Ta" and followed by exactly one character.



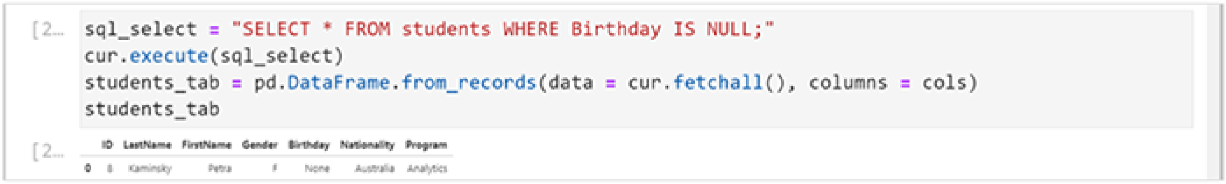
```
[2… sql_select = "SELECT * FROM students WHERE Program = 'Analytics' AND LastName LIKE 'Ta_';"
    cur.execute(sql_select)
    students_tab = pd.DataFrame.from_records(data = cur.fetchall(), columns = cols)
    students_tab
```

```
[2…    ID  LastName  FirstName  Gender     Birthday  Nationality   Program
    0   1       Tan      Peter       M   2002-08-13    Singapore  Analytics
    1   5       Tay      Susan       M   2002-07-12    Singapore  Analytics
    2  19       Tan    Clement       M   2002-03-11    Singapore  Analytics
```

Figure 6.25 Select Analytics Students with Last Names that Have the Pattern "Ta_"

The syntax in Figure 6.25 will result in selecting all last names with three characters that start with "Ta". In other words, last names such as "Tang" will not be included.

In Figure 6.26, cases with missing value in `Birthday` will be selected. This step is usually helpful to let data analysts study the cases with missing values first and then decide to remove them from the dataset or not.

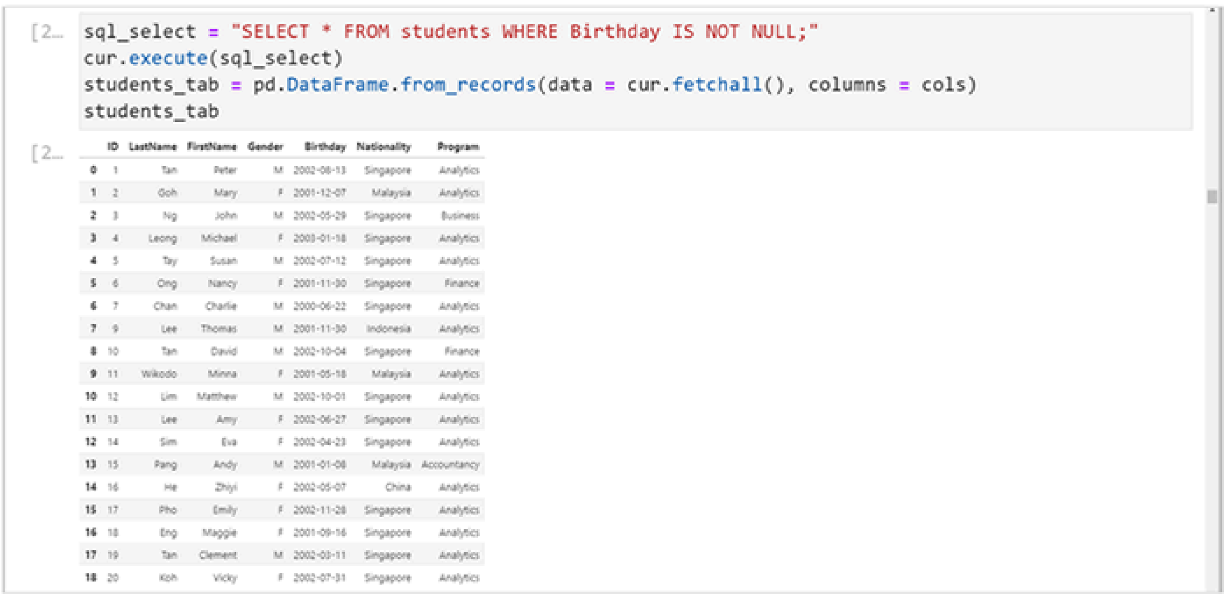


```
[2… sql_select = "SELECT * FROM students WHERE Birthday IS NULL;"
    cur.execute(sql_select)
    students_tab = pd.DataFrame.from_records(data = cur.fetchall(), columns = cols)
    students_tab
```

```
[2…    ID  LastName  FirstName  Gender  Birthday  Nationality   Program
    0   8  Kaminsky      Petra       F      None    Australia  Analytics
```

Figure 6.26 Select Students with No Birthday Record

If we simply want to select all students whose birthday records are not missing, we will just need to replace `IS NULL` by `IS NOT NULL` in the `SELECT` statement.



```
[2…  sql_select = "SELECT * FROM students WHERE Birthday IS NOT NULL;"
     cur.execute(sql_select)
     students_tab = pd.DataFrame.from_records(data = cur.fetchall(), columns = cols)
     students_tab
```

[2…

	ID	LastName	FirstName	Gender	Birthday	Nationality	Program
0	1	Tan	Peter	M	2002-08-13	Singapore	Analytics
1	2	Goh	Mary	F	2001-12-07	Malaysia	Analytics
2	3	Ng	John	M	2002-05-29	Singapore	Business
3	4	Leong	Michael	F	2003-01-18	Singapore	Analytics
4	5	Tay	Susan	M	2002-07-12	Singapore	Analytics
5	6	Ong	Nancy	F	2001-11-30	Singapore	Finance
6	7	Chan	Charlie	M	2000-06-22	Singapore	Analytics
7	9	Lee	Thomas	M	2001-11-30	Indonesia	Analytics
8	10	Tan	David	M	2002-10-04	Singapore	Finance
9	11	Wikodo	Minna	F	2001-05-18	Malaysia	Analytics
10	12	Lim	Matthew	M	2002-10-01	Singapore	Analytics
11	13	Lee	Amy	F	2002-06-27	Singapore	Analytics
12	14	Sim	Eva	F	2002-04-23	Singapore	Analytics
13	15	Pang	Andy	M	2001-01-08	Malaysia	Accountancy
14	16	He	Zhiyi	F	2002-05-07	China	Analytics
15	17	Pho	Emily	F	2002-11-28	Singapore	Analytics
16	18	Eng	Maggie	F	2001-09-16	Singapore	Analytics
17	19	Tan	Clement	M	2002-03-11	Singapore	Analytics
18	20	Koh	Vicky	F	2002-07-31	Singapore	Analytics

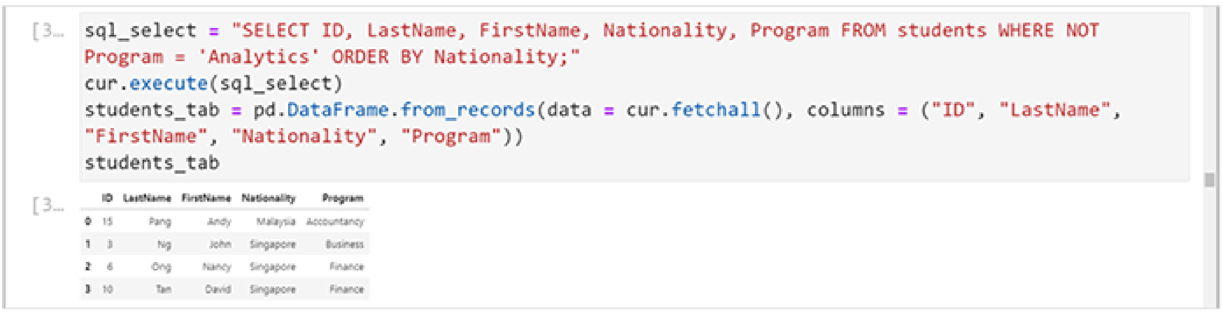
Figure 6.27 Select Students with Non-Missing Birthday Records

In SQL, we can also select particular columns from a table in the data query. The asterisk (`*`) in the `SELECT` statement should be replaced by a list of selected variables in this case.

```
SELECT var_name1, var_name2, …
FROM table_name WHERE criteria;
```

It is also possible to use Python programming to manipulate the `SELECT` statement as string before sending it to SQL. That is, we can create our own variable list as string in our Python program first and then combine it with the rest of the statement. This approach will give us the flexibility to generate different variable lists for data query depending on the requirement of the situation.

Example (Cont'd): In Figure 6.28, we select the columns `ID`, `LastName`, `FirstName`, `Nationality` and `Program` from the table `students`. At the same time, we are only interested in students outside the analytics program. However, these records should also be sorted by their nationalities.

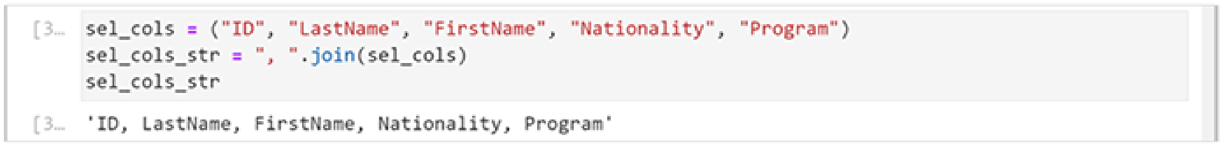


```
[3… sql_select = "SELECT ID, LastName, FirstName, Nationality, Program FROM students WHERE NOT
    Program = 'Analytics' ORDER BY Nationality;"
    cur.execute(sql_select)
    students_tab = pd.DataFrame.from_records(data = cur.fetchall(), columns = ("ID", "LastName",
    "FirstName", "Nationality", "Program"))
    students_tab
```

```
[3…    ID  LastName  FirstName  Nationality     Program
    0  15      Pang       Andy     Malaysia  Accountancy
    1   3        Ng       John    Singapore     Business
    2   6       Ong      Nancy    Singapore      Finance
    3  10       Tan      David    Singapore      Finance
```

Figure 6.28 Select Data of Non-Analytics Students from Certain Columns of a Table

The parameter `columns` in the `from_records()` function must be replaced by a new variable list since the list `cols` which we have generated in Figure 6.13 contains more column names than the table we query here. As a result, it is more practical to store the names of the selected variable in a list or a tuple first, and then concatenate them to a single string by the `join()` method.



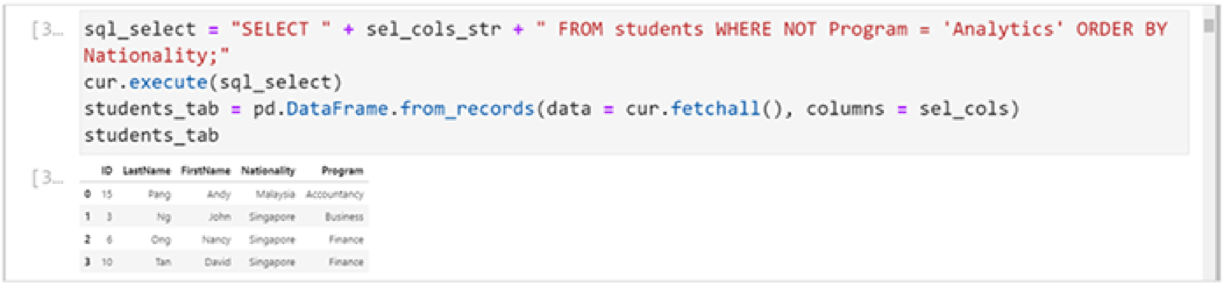
```
[3… sel_cols = ("ID", "LastName", "FirstName", "Nationality", "Program")
    sel_cols_str = ", ".join(sel_cols)
    sel_cols_str
```

```
[3… 'ID, LastName, FirstName, Nationality, Program'
```

Figure 6.29 Concatenate Items of a Tuple to a String

In the first line, we define the tuple named `sel_cols` with the names of the selected variables in it. In the second line, the `.join()` method is applied to `", "`, the separator between the variable names in the output string. Basically, the `.join()` method runs through all the items in `sel_cols` and adds them with `", "` (except for the last item for which the separator is not necessary) to the string `sel_cols_str` one after another.

In the next step, we first split the `SELECT` statement in Figure 6.28 into three parts: `"Select "`, the variable list and the rest. The original variable list can then be replaced by the string variable `sel_cols_str`. Subsequently, we can concatenate these three partitions of the string into one by "adding" them together as shown in Figure 6.30. Furthermore, we can also use the variable list `sel_cols` as the value for the parameter `columns` in the `from_records()` function.



```
[3… sql_select = "SELECT " + sel_cols_str + " FROM students WHERE NOT Program = 'Analytics' ORDER BY
    Nationality;"
    cur.execute(sql_select)
    students_tab = pd.DataFrame.from_records(data = cur.fetchall(), columns = sel_cols)
    students_tab
```

[3…

	ID	LastName	FirstName	Nationality	Program
0	15	Pang	Andy	Malaysia	Accountancy
1	3	Ng	John	Singapore	Business
2	6	Ong	Nancy	Singapore	Finance
3	10	Tan	David	Singapore	Finance

Figure 6.30 Select Certain Columns of a Table by a Variable List String

Hereafter, we can obtain different columns of the table by just changing the variable names in `sel_cols_str` and re-run the code in Figure 6.30. Note that the same technique can also be applied to the other parts of the `SELECT` statement.

Read

Refer to the links below for more details and examples on the `WHERE` clause of SQL:

https://www.w3schools.com/sql/sql_where.asp

https://www.sqlitetutorial.net/sqlite-where/

Refer to the link below for more details and examples on the `AND`, `OR`, `NOT` operators of SQL:

https://www.w3schools.com/sql/sql_and_or.asp

Refer to the links below for more details and examples on the `IN` operator of SQL:

https://www.w3schools.com/sql/sql_in.asp

https://www.sqlitetutorial.net/sqlite-in/

Refer to the links below for more details and examples on the `BETWEEN` operator of SQL:

https://www.w3schools.com/sql/sql_between.asp

https://www.sqlitetutorial.net/sqlite-between/

Refer to the links below for more details and examples on the `LIKE` operator of SQL:

https://www.w3schools.com/sql/sql_like.asp

https://www.sqlitetutorial.net/sqlite-like/

Refer to the link below for more details and examples on the wildcard characters in SQL:

https://www.w3schools.com/sql/sql_wildcards.asp

Refer to the links below for more details and examples on `NULL` values in SQL:

https://www.w3schools.com/sql/sql_null_values.asp

https://www.sqlitetutorial.net/sqlite-is-null/

Refer to the link below for more details and examples on the `.join()` methods:

https://docs.python.org/3/library/stdtypes.html#str.join

Chapter 3: Joining Tables

Lesson Recording

Join Tables with SQL

The tables in a database are usually connected in some ways. For instance, in the database of a bank, there may be a table with the personal records of all the customer relationship managers and another table with the records of all customers, including their transaction records and the names of their relationship manager. With these data, the bank can query on the sales records of each manager within a certain period. In this case, we are joining tables from a database to gain cross-table information.

3.1 Inner Join

In SQL, there are many ways to join two or more tables: `INNER JOIN`, `LEFT JOIN`, `CROSS JOIN`, etc. Depending on the structure of the tables, these join techniques usually result in different output tables. We will discuss the inner join method in this section.

```
SELECT * FROM table1_name
INNER JOIN table2_name
ON table1_name.match_var = table2_name.match_var;
```

The `INNER JOIN` clause is used within the `SELECT` statement. It selects only records of `table1` that can be matched by records in `table2`. The rows of `table1` or `table2` for which SQL cannot find any matches in the opposite table will be dropped from the query. SQL compares the values of each one matching variable from the two tables specified by the user. Usually, these variables should represent the same feature in both tables such as the employee number or customer ID. Note that it is also possible to extend the match to multiple pairs of variables if necessary.

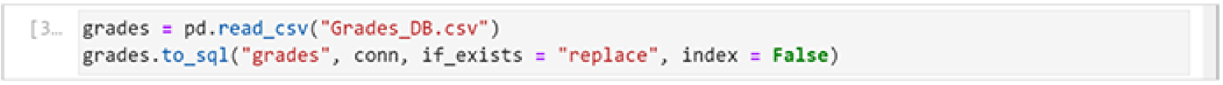
The matching condition should be provided after the `ON` keyword. To indicate the original table of the matching variables, the name of the table must be specified before each matching variable and separated by a dot (`.`). With the `ON` keyword, the matching variables do not need to have the same name in their original tables. But if they do, we can shorten the syntax above by the `USING` keyword.

```
SELECT * FROM table1_name
INNER JOIN table2_name USING(match_var);
```

The name of the original table does not need to be mentioned in the bracket of the `USING` keyword. If, however, the column name only exists in one of the tables, SQL will return an error message to us. Another difference between the `ON` and `USING` keywords is that both matching variables will be included in the output table if we use the `ON` keyword, whereas only the matching variable of `table1` will remain if the `USING` keyword is used for matching.

If more than one records in `table2` are found matching to a record in `table1`, SQL will append each of them to a copy of the matched record of `table1`. This is the so-called 1:n matching.

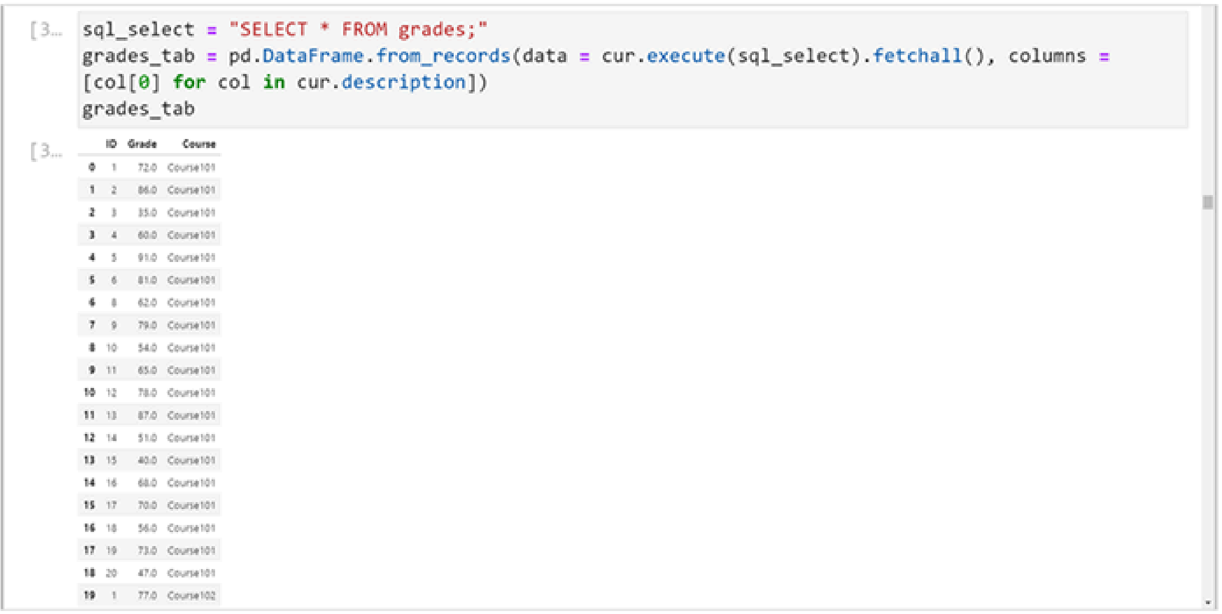
Example (Cont'd): In the following, we will create a new table called `grades` which contains the scores of the students in the two previous examinations. The table will then be joined with the table `students`. In Figure 6.31, the data stored in a .csv file will be transferred to `grades` by the `.to_sql()` function.



```
[3…  grades = pd.read_csv("Grades_DB.csv")
     grades.to_sql("grades", conn, if_exists = "replace", index = False)
```

Figure 6.31 Create a New Table Called `grades` in the Database

In Figure 6.32, we query the entire table `grades` from the database and convert it to a pandas Database for printing and controlling purposes.

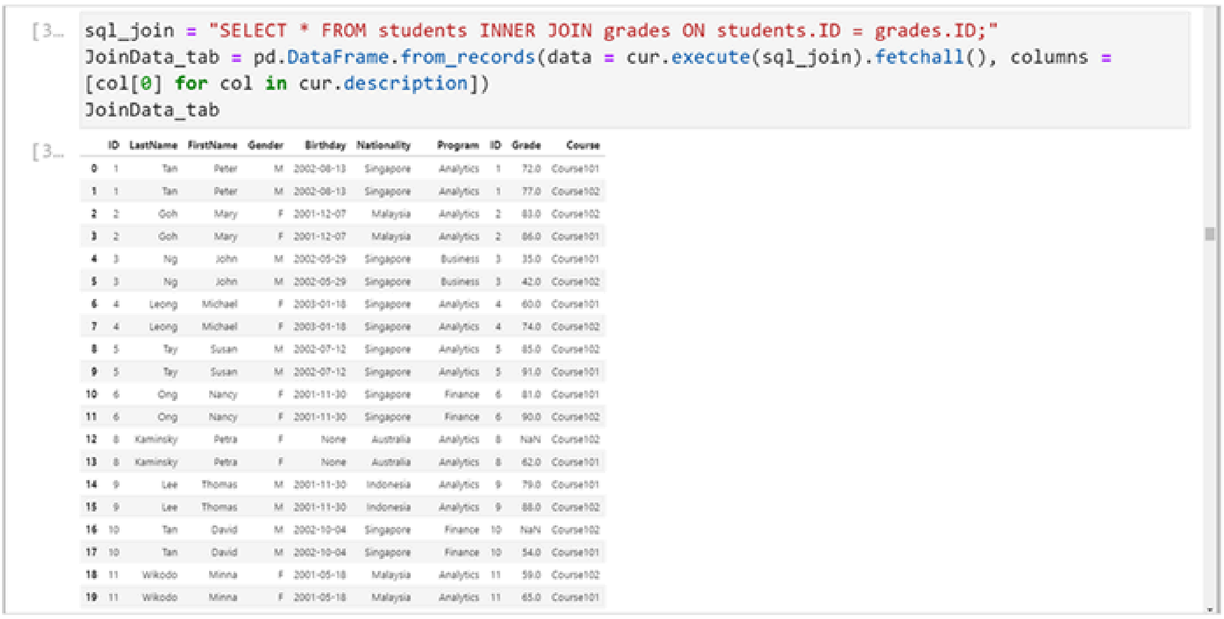


```
[3…  sql_select = "SELECT * FROM grades;"
     grades_tab = pd.DataFrame.from_records(data = cur.execute(sql_select).fetchall(), columns =
     [col[0] for col in cur.description])
     grades_tab
```

```
[3…      ID  Grade     Course
     0    1   72.0  Course101
     1    2   86.0  Course101
     2    3   35.0  Course101
     3    4   60.0  Course101
     4    5   91.0  Course101
     5    6   81.0  Course101
     6    8   62.0  Course101
     7    9   79.0  Course101
     8   10   54.0  Course101
     9   11   65.0  Course101
    10   12   78.0  Course101
    11   13   87.0  Course101
    12   14   51.0  Course101
    13   15   40.0  Course101
    14   16   68.0  Course101
    15   17   70.0  Course101
    16   18   56.0  Course101
    17   19   73.0  Course101
    18   20   47.0  Course101
    19    1   77.0  Course102
```

Figure 6.32 Contents of the Table `grades`

Subsequently, we join the two tables, `students` and `grades`, with the `INNER JOIN` clause. In our first attempt, we use the `ON` keyword to match the student ID of the two tables.



```
[3… sql_join = "SELECT * FROM students INNER JOIN grades ON students.ID = grades.ID;"
    JoinData_tab = pd.DataFrame.from_records(data = cur.execute(sql_join).fetchall(), columns =
    [col[0] for col in cur.description])
    JoinData_tab
```

[3…

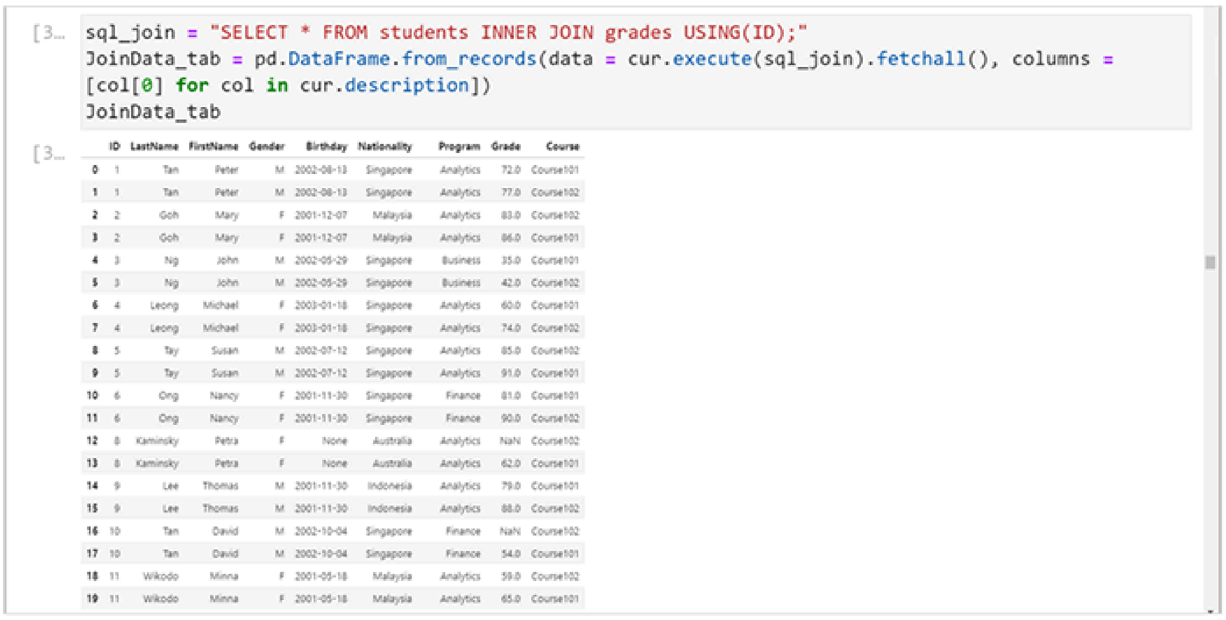
	ID	LastName	FirstName	Gender	Birthday	Nationality	Program	ID	Grade	Course
0	1	Tan	Peter	M	2002-08-13	Singapore	Analytics	1	72.0	Course101
1	1	Tan	Peter	M	2002-08-13	Singapore	Analytics	1	77.0	Course102
2	2	Goh	Mary	F	2001-12-07	Malaysia	Analytics	2	83.0	Course102
3	2	Goh	Mary	F	2001-12-07	Malaysia	Analytics	2	86.0	Course101
4	3	Ng	John	M	2002-05-29	Singapore	Business	3	35.0	Course101
5	3	Ng	John	M	2002-05-29	Singapore	Business	3	42.0	Course102
6	4	Leong	Michael	F	2003-01-18	Singapore	Analytics	4	60.0	Course101
7	4	Leong	Michael	F	2003-01-18	Singapore	Analytics	4	74.0	Course102
8	5	Tay	Susan	M	2002-07-12	Singapore	Analytics	5	85.0	Course102
9	5	Tay	Susan	M	2002-07-12	Singapore	Analytics	5	91.0	Course101
10	6	Ong	Nancy	F	2001-11-30	Singapore	Finance	6	81.0	Course101
11	6	Ong	Nancy	F	2001-11-30	Singapore	Finance	6	90.0	Course102
12	8	Kaminsky	Petra	F	None	Australia	Analytics	8	NaN	Course102
13	8	Kaminsky	Petra	F	None	Australia	Analytics	8	62.0	Course101
14	9	Lee	Thomas	M	2001-11-30	Indonesia	Analytics	9	79.0	Course101
15	9	Lee	Thomas	M	2001-11-30	Indonesia	Analytics	9	88.0	Course102
16	10	Tan	David	M	2002-10-04	Singapore	Finance	10	NaN	Course102
17	10	Tan	David	M	2002-10-04	Singapore	Finance	10	54.0	Course101
18	11	Wikodo	Minna	F	2001-05-18	Malaysia	Analytics	11	59.0	Course102
19	11	Wikodo	Minna	F	2001-05-18	Malaysia	Analytics	11	65.0	Course101

Figure 6.33 Inner Join the Tables `students` and `grades` by the `ON` Keyword

We can see from Figure 6.33 that most of the students have two rows in the output table. Since the majority of them have taken part in both examinations, they should have indeed two records each in the table `grades`. With the `INNER JOIN` clause, SQL generates one copy of each student's record in `students` and appends the scores and course codes found in `grades` to the records of the corresponding student.

Furthermore, we can also see from Figure 6.33 that there are two columns named `ID`. As mentioned above, the matching variables of both tables will be kept in the output table. If we intended to use this table for further process, the ambiguous variable name could cause troubles in data query. It would therefore be important to either rename or drop one of the `ID` columns from the table.

Since the matching variable in both tables share the same name, `ID`, we can also apply the `USING` keyword instead.



```
[3…  sql_join = "SELECT * FROM students INNER JOIN grades USING(ID);"
     JoinData_tab = pd.DataFrame.from_records(data = cur.execute(sql_join).fetchall(), columns =
     [col[0] for col in cur.description])
     JoinData_tab
```

```
[3…      ID  LastName  FirstName  Gender    Birthday  Nationality    Program  Grade      Course
     0    1       Tan      Peter       M  2002-08-13    Singapore  Analytics   72.0   Course101
     1    1       Tan      Peter       M  2002-08-13    Singapore  Analytics   77.0   Course102
     2    2       Goh       Mary       F  2001-12-07     Malaysia  Analytics   83.0   Course102
     3    2       Goh       Mary       F  2001-12-07     Malaysia  Analytics   86.0   Course101
     4    3        Ng       John       M  2002-05-29    Singapore   Business   35.0   Course101
     5    3        Ng       John       M  2002-05-29    Singapore   Business   42.0   Course102
     6    4     Leong    Michael       F  2003-01-18    Singapore  Analytics   60.0   Course101
     7    4     Leong    Michael       F  2003-01-18    Singapore  Analytics   74.0   Course102
     8    5       Tay      Susan       M  2002-07-12    Singapore  Analytics   85.0   Course102
     9    5       Tay      Susan       M  2002-07-12    Singapore  Analytics   91.0   Course101
    10    6       Ong      Nancy       F  2001-11-30    Singapore    Finance   81.0   Course101
    11    6       Ong      Nancy       F  2001-11-30    Singapore    Finance   90.0   Course102
    12    8  Kaminsky      Petra       F        None    Australia  Analytics    NaN   Course102
    13    8  Kaminsky      Petra       F        None    Australia  Analytics   62.0   Course101
    14    9       Lee     Thomas       M  2001-11-30    Indonesia  Analytics   79.0   Course101
    15    9       Lee     Thomas       M  2001-11-30    Indonesia  Analytics   88.0   Course102
    16   10       Tan      David       M  2002-10-04    Singapore    Finance    NaN   Course102
    17   10       Tan      David       M  2002-10-04    Singapore    Finance   54.0   Course101
    18   11    Wikodo      Minna       F  2001-05-18     Malaysia  Analytics   59.0   Course102
    19   11    Wikodo      Minna       F  2001-05-18     Malaysia  Analytics   65.0   Course101
```

Figure 6.34 Inner Join the Tables `students` and `grades` by the `USING` Keyword

Read

Refer to the links below for more details and examples on the `INNER JOIN` clause of SQL:

https://www.w3schools.com/sql/sql_join_inner.asp

https://www.sqlitetutorial.net/sqlite-inner-join/

3.2 Left Join

Another way to join two or more tables of a database is the left join method.

```
SELECT * FROM table1_name alias1
LEFT JOIN table2_name alias2
ON alias1.match_var = alias2.match_var;
```

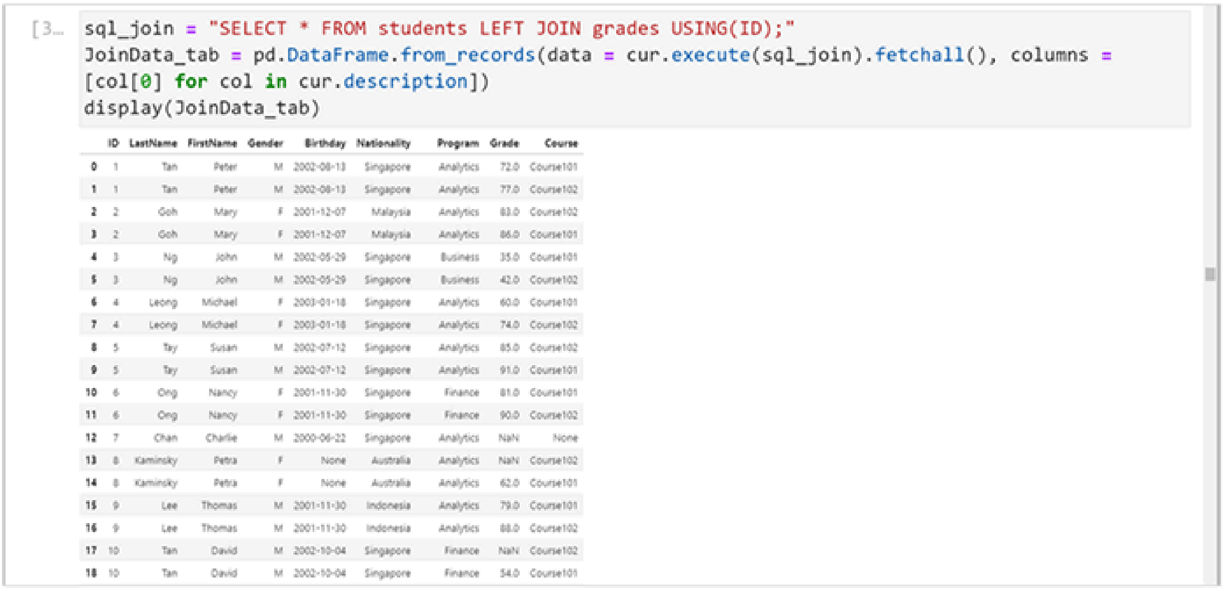
We add the new options `alias1` and `alias2` to this syntax. These aliases are usually abbreviated references of `table1` and `table2` and can be useful when there are two variables with the same name in both tables. Note that the aliases are not limited to the `LEFT JOIN` clause, but applicable to any `SELECT` statement.

We can also use `USING` instead of `ON` for the matching condition.

```
SELECT * FROM table1_name
LEFT JOIN table2_name
USING(match_var);
```

In left join, SQL searches for records from `table2` that match the records from `table1` based on the matching condition. If no records in `table2` can be found for a record from `table1`, that record of `table1` will still be kept in the output table. The values of the variables originated from `table2` will be missing values in this case.

Example (Cont'd): The tables `students` and `grades` will be merged by the `LEFT JOIN` method in the following.



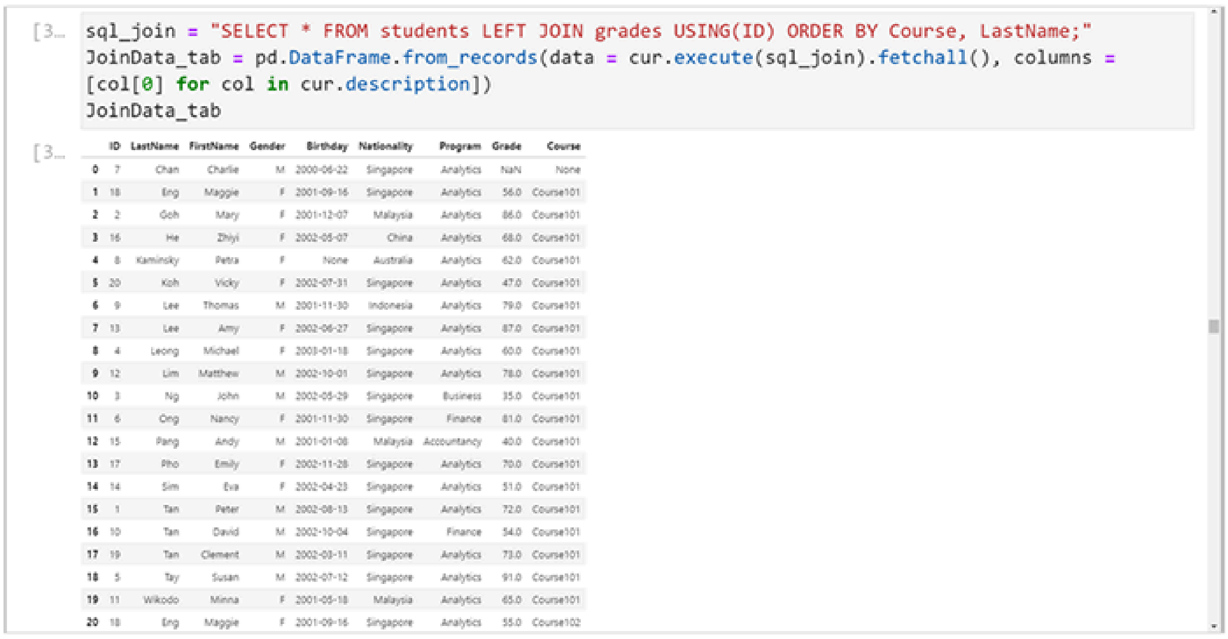
```
[3… sql_join = "SELECT * FROM students LEFT JOIN grades USING(ID);"
    JoinData_tab = pd.DataFrame.from_records(data = cur.execute(sql_join).fetchall(), columns =
    [col[0] for col in cur.description])
    display(JoinData_tab)
```

	ID	LastName	FirstName	Gender	Birthday	Nationality	Program	Grade	Course
0	1	Tan	Peter	M	2002-08-13	Singapore	Analytics	72.0	Course101
1	1	Tan	Peter	M	2002-08-13	Singapore	Analytics	77.0	Course102
2	2	Goh	Mary	F	2001-12-07	Malaysia	Analytics	83.0	Course102
3	2	Goh	Mary	F	2001-12-07	Malaysia	Analytics	86.0	Course101
4	3	Ng	John	M	2002-05-29	Singapore	Business	35.0	Course101
5	3	Ng	John	M	2002-05-29	Singapore	Business	42.0	Course102
6	4	Leong	Michael	F	2003-01-18	Singapore	Analytics	60.0	Course101
7	4	Leong	Michael	F	2003-01-18	Singapore	Analytics	74.0	Course102
8	5	Tay	Susan	M	2002-07-12	Singapore	Analytics	85.0	Course102
9	5	Tay	Susan	M	2002-07-12	Singapore	Analytics	91.0	Course101
10	6	Ong	Nancy	F	2001-11-30	Singapore	Finance	81.0	Course101
11	6	Ong	Nancy	F	2001-11-30	Singapore	Finance	90.0	Course102
12	7	Chan	Charlie	M	2000-06-22	Singapore	Analytics	NaN	None
13	8	Kaminsky	Petra	F	None	Australia	Analytics	NaN	Course102
14	8	Kaminsky	Petra	F	None	Australia	Analytics	62.0	Course101
15	9	Lee	Thomas	M	2001-11-30	Indonesia	Analytics	79.0	Course101
16	9	Lee	Thomas	M	2001-11-30	Indonesia	Analytics	88.0	Course102
17	10	Tan	David	M	2002-10-04	Singapore	Finance	NaN	Course102
18	10	Tan	David	M	2002-10-04	Singapore	Finance	54.0	Course101

Figure 6.35 Left Join the Tables `students` and `grades`

In Figure 6.35, we can see that the student with ID number 7 has missing data in the variables `Grade` and `Course`, both variables originated from the table `grades`. We can therefore conclude that this student has not taken part in both the examinations of Course101 and Course102.

We can also sort the joined table by some selected variables. All we need to do is to add the `ORDER BY` keyword to the `SELECT` statement as shown in Figure 6.36.



```
[3… sql_join = "SELECT * FROM students LEFT JOIN grades USING(ID) ORDER BY Course, LastName;"
    JoinData_tab = pd.DataFrame.from_records(data = cur.execute(sql_join).fetchall(), columns =
    [col[0] for col in cur.description])
    JoinData_tab
```

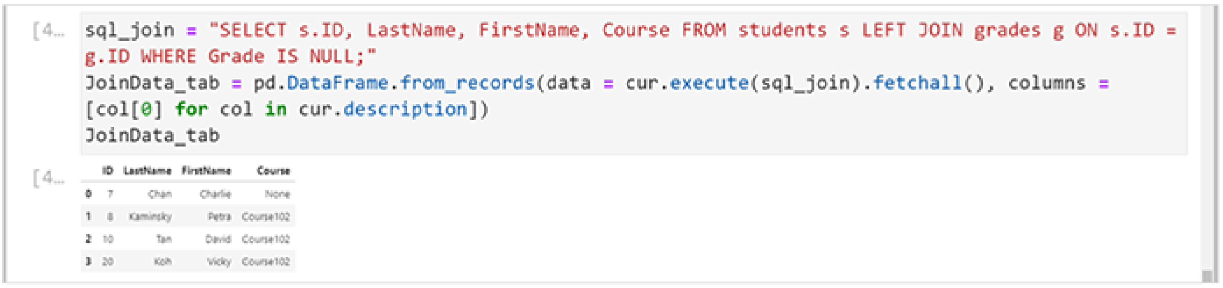
[3…

	ID	LastName	FirstName	Gender	Birthday	Nationality	Program	Grade	Course
0	7	Chan	Charlie	M	2000-06-22	Singapore	Analytics	NaN	None
1	18	Eng	Maggie	F	2001-09-16	Singapore	Analytics	56.0	Course101
2	2	Goh	Mary	F	2001-12-07	Malaysia	Analytics	86.0	Course101
3	16	He	Zhiyi	F	2002-05-07	China	Analytics	68.0	Course101
4	8	Kaminsky	Petra	F	None	Australia	Analytics	62.0	Course101
5	20	Koh	Vicky	F	2002-07-31	Singapore	Analytics	47.0	Course101
6	9	Lee	Thomas	M	2001-11-30	Indonesia	Analytics	79.0	Course101
7	13	Lee	Amy	F	2002-06-27	Singapore	Analytics	87.0	Course101
8	4	Leong	Michael	F	2003-01-18	Singapore	Analytics	60.0	Course101
9	12	Lim	Matthew	M	2002-10-01	Singapore	Analytics	78.0	Course101
10	3	Ng	John	M	2002-05-29	Singapore	Business	35.0	Course101
11	6	Ong	Nancy	F	2001-11-30	Singapore	Finance	81.0	Course101
12	15	Pang	Andy	M	2001-01-08	Malaysia	Accountancy	40.0	Course101
13	17	Pho	Emily	F	2002-11-28	Singapore	Analytics	70.0	Course101
14	14	Sim	Eva	F	2002-04-23	Singapore	Analytics	51.0	Course101
15	1	Tan	Peter	M	2002-08-13	Singapore	Analytics	72.0	Course101
16	10	Tan	David	M	2002-10-04	Singapore	Finance	54.0	Course101
17	19	Tan	Clement	M	2002-03-11	Singapore	Analytics	73.0	Course101
18	5	Tay	Susan	M	2002-07-12	Singapore	Analytics	91.0	Course101
19	11	Wikodo	Minna	F	2001-05-18	Malaysia	Analytics	65.0	Course101
20	18	Eng	Maggie	F	2001-09-16	Singapore	Analytics	55.0	Course102

Figure 6.36 Sort the Left Joined Table by the Course Code and the Students' Last Name

In Figure 6.36, we have sorted the output table by the course code and the last name of the students, both in ascending order. By sorting the data this way, we can see the examination scores of one course as consecutive records, and students from the same course are sorted in the same order as most probably the official student list, namely by their last names in alphabetical order.

In Figure 6.37, all records from the joined table that have missing values in the variable `Grade` are selected.



```
[4… sql_join = "SELECT s.ID, LastName, FirstName, Course FROM students s LEFT JOIN grades g ON s.ID =
    g.ID WHERE Grade IS NULL;"
    JoinData_tab = pd.DataFrame.from_records(data = cur.execute(sql_join).fetchall(), columns =
    [col[0] for col in cur.description])
    JoinData_tab
```

[4…

	ID	LastName	FirstName	Course
0	7	Chan	Charlie	None
1	8	Kaminsky	Petra	Course102
2	10	Tan	David	Course102
3	20	Koh	Vicky	Course102

Figure 6.37 Select Records with Missing Data in `grade`

We add the aliases `s` for the table `students` and `g` for the table `grades` to the `SELECT` statement. They are included in the matching criterion since the `ON` keyword is used here for record matching. Furthermore, we just want the joined table to contain the columns `ID`, `LastName`, `FirstName` and `Course`. Since there is an `ID` column in both tables and only the one from the table `students` should be carried over to the output table, we need to specify its original table with the alias `s`.

Read

Refer to the links below for more details and examples on the `LEFT JOIN` clause of SQL:

https://www.w3schools.com/sql/sql_join_left.asp

https://www.sqlitetutorial.net/sqlite-left-join/

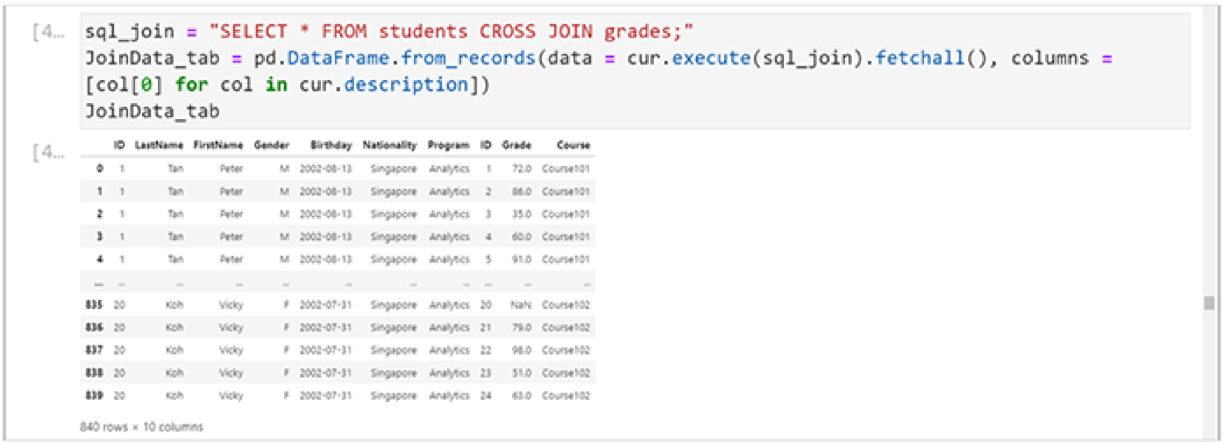
3.3 Cross Join

With the cross join method, SQL produces the cartesian product of the involved tables.

```
SELECT * FROM table1_name
CROSS JOIN table2_name;
```

The cartesian product usually refers to the collection of all cross-item combinations resulting from two arrays. In terms of the cross join method, it means that every record of `table1` is merged with *all* records of `table2`. In other words, if `table1` and `table2` have m and n records, respectively, there will be a total of m×n records in the output table. Since no matches are required here, the `ON` and `USING` keywords can be omitted in the `SELECT` statement.

Example (Cont'd): Now we cross join the tables `students` and `grades`.



```
[4... sql_join = "SELECT * FROM students CROSS JOIN grades;"
      JoinData_tab = pd.DataFrame.from_records(data = cur.execute(sql_join).fetchall(), columns =
      [col[0] for col in cur.description])
      JoinData_tab
```

```
[4...      ID  LastName  FirstName  Gender     Birthday  Nationality   Program  ID  Grade      Course
      0     1       Tan      Peter       M   2002-08-13    Singapore  Analytics   1   72.0   Course101
      1     1       Tan      Peter       M   2002-08-13    Singapore  Analytics   2   86.0   Course101
      2     1       Tan      Peter       M   2002-08-13    Singapore  Analytics   3   35.0   Course101
      3     1       Tan      Peter       M   2002-08-13    Singapore  Analytics   4   60.0   Course101
      4     1       Tan      Peter       M   2002-08-13    Singapore  Analytics   5   91.0   Course101
    ...   ...       ...        ...     ...          ...          ...        ...  ...    ...         ...
    835    20       Koh      Vicky       F   2002-07-31    Singapore  Analytics  20    NaN   Course102
    836    20       Koh      Vicky       F   2002-07-31    Singapore  Analytics  21   79.0   Course102
    837    20       Koh      Vicky       F   2002-07-31    Singapore  Analytics  22   98.0   Course102
    838    20       Koh      Vicky       F   2002-07-31    Singapore  Analytics  23   51.0   Course102
    839    20       Koh      Vicky       F   2002-07-31    Singapore  Analytics  24   63.0   Course102

    840 rows × 10 columns
```

Figure 6.38 Cross Join the Tables `students` and `grades`

Every row of the table `students` is now cross-combined with all the rows of the table `grades`. In other words, the output table contains records in which students are assigned to grades that they did not score themselves. It is obvious that this data query does not make much sense in terms of its logical structure and analytical value. Nevertheless, cross join could be useful if we had another table with data that are equal to every student. We could then merge the same records to every row of the table `students`.

Read

Refer to the link below for more details and examples on the `CROSS JOIN` clause of SQL:

https://www.sqlitetutorial.net/sqlite-cross-join/

3.4 Outer Join

With the outer join method, or full outer join, SQL produces the union of the involved tables. In other words, not only records from both tables that can be matched by the matching criterion will be selected, records from either one table that cannot find any match from the opposite side will also be carried over in the output table. However, their values in the variables originated from the other tables will be `None`.

One difficulty of applying the outer join method in Python is the fact that it is simply not supported by the sqlite3 package, although the `OUTER JOIN` clause is actually available in other SQL versions. Nevertheless, we can combine the `LEFT JOIN` clause with the `UNION ALL` operator to create the same result as the `OUTER JOIN` clause.

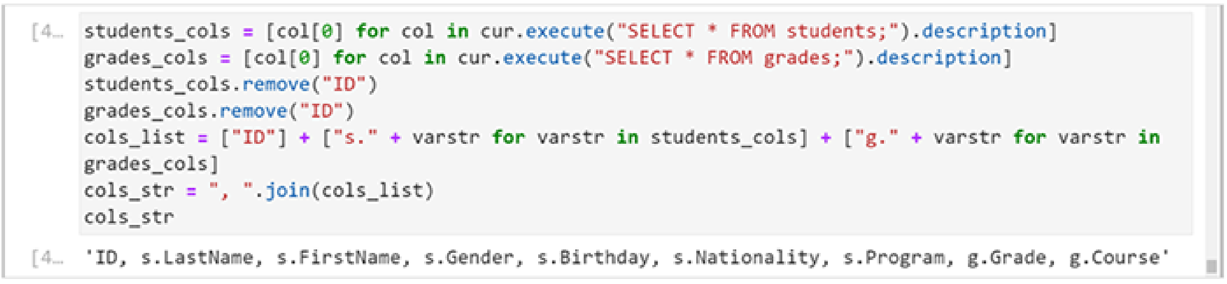
```
SELECT var_list FROM table1_name alias1
LEFT JOIN table2_name alias2 USING(matching_var)
UNION ALL
SELECT var_list FROM table2_name alias2
LEFT JOIN table1_name alias1 USING(matching_var)
WHERE alias1.var_name IS NULL;
```

In the first `SELECT` statement, the left join method is applied and the records from `table2` are matched with the records from `table1`. As discussed in Chapter 3.2, all records from `table1` are selected in the output table here regardless the matching results. In the second `SELECT` statement, the left join method is applied to `table1` and `table2` in the opposite roles. As a result, all records from `table2` are selected here. However, we need to drop those matches that are already included in the first `SELECT` statement to prevent duplicates. Logically, these records must contain data from both tables, and those from `table2` with no matching records must have missing data in the columns of `table1`. As a result, we can simply use this result as our selection criterion. Both `SELECT` statements are connected by the `UNION ALL` operator to produce a combined table of the two queries.

Since the sequence of the columns are naturally different in the output tables of the two queries, the `UNION ALL` operator would simply append the data of the second query to those of the first query regardless their original columns if we just used the asterisk (`*`) in both `SELECT` statements. To avoid such mess in the output data, we must specify the same list of variable names in both queries so that the sequence of the columns are identical.

Example (Cont'd): Before we use outer join to merge the tables `students` and `grades`, we need to create a list with the variable names of both tables to ensure that the same sequence of columns are produced in the output table of both queries.

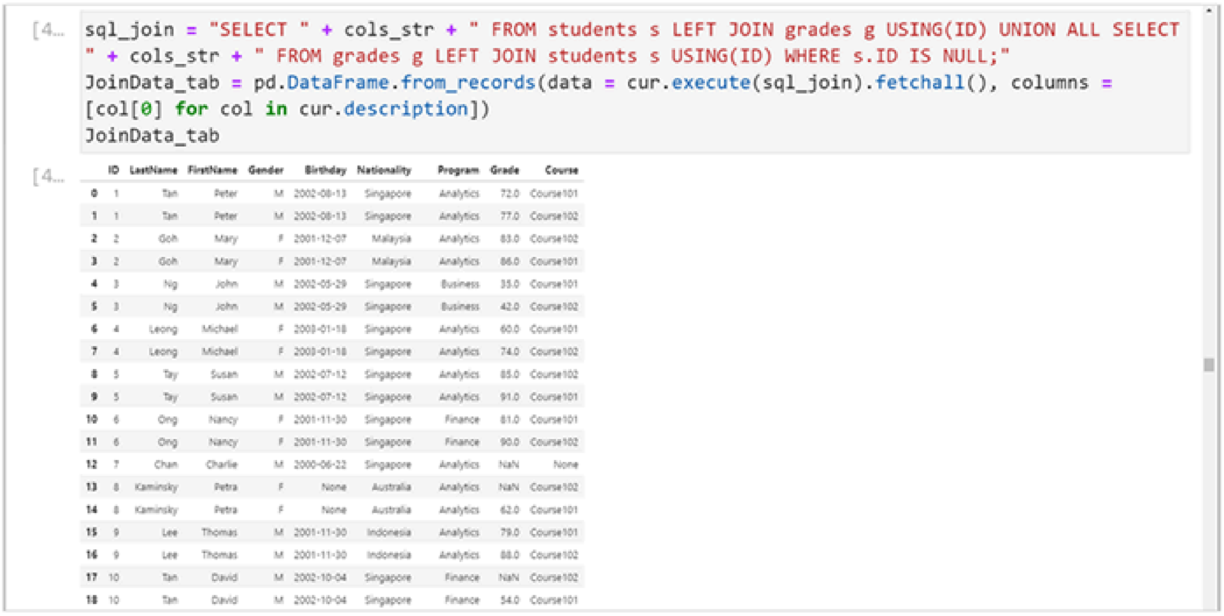
In the column list, we must specify the original table of each variable by the aliases so that every column in the output table is uniquely defined. The easiest way to generate such a list is to define a string with all the variable names written in it. However, if the involved tables have many columns, we will need to write a very long string in our program which is rather not efficient. Figure 6.39 shows a Python program which generates a variable list from the column names of both tables together with the corresponding aliases.



```
[4… students_cols = [col[0] for col in cur.execute("SELECT * FROM students;").description]
    grades_cols = [col[0] for col in cur.execute("SELECT * FROM grades;").description]
    students_cols.remove("ID")
    grades_cols.remove("ID")
    cols_list = ["ID"] + ["s." + varstr for varstr in students_cols] + ["g." + varstr for varstr in
    grades_cols]
    cols_str = ", ".join(cols_list)
    cols_str

[4… 'ID, s.LastName, s.FirstName, s.Gender, s.Birthday, s.Nationality, s.Program, g.Grade, g.Course'
```

Figure 6.39 Generate Variable List from Both Tables with Aliases

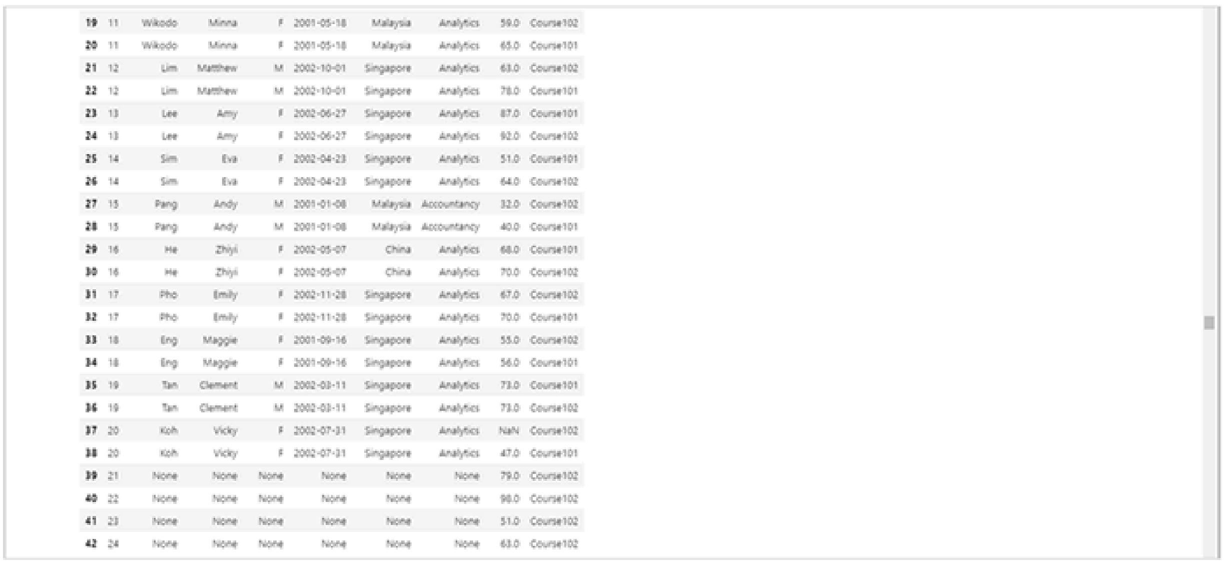
In the first two lines, the column names of the tables `students` and `grades` are stored in the lists `students_cols` and `grades_cols` by the same technique as shown in Figure 6.13. In the third and fourth line, we remove both `ID` variables from the lists since they will be merged into one column eventually and therefore do not need aliases. In the fifth line, we first add the aliases `"s."` and `"g."` as strings to each item of the respective variable list by running through them in a `for`-loop. Subsequently, the two lists are concatenated, and the item `"ID"` is added back to the front of the resulting list which is now called `cols_list`. In line six, `cols_list` is converted to a string with commas separating the column names. As shown in Figure 6.39, an alias is added to all names except for the `ID` variable, which has now become the first column of the output tables in both `SELECT` statements.



```
[4...  sql_join = "SELECT " + cols_str + " FROM students s LEFT JOIN grades g USING(ID) UNION ALL SELECT
       " + cols_str + " FROM grades g LEFT JOIN students s USING(ID) WHERE s.ID IS NULL;"
       JoinData_tab = pd.DataFrame.from_records(data = cur.execute(sql_join).fetchall(), columns =
       [col[0] for col in cur.description])
       JoinData_tab
```

```
[4...    ID  LastName  FirstName  Gender    Birthday  Nationality    Program  Grade      Course
      0   1       Tan      Peter       M  2002-08-13    Singapore  Analytics   72.0  Course101
      1   1       Tan      Peter       M  2002-08-13    Singapore  Analytics   77.0  Course102
      2   2       Goh       Mary       F  2001-12-07     Malaysia  Analytics   83.0  Course102
      3   2       Goh       Mary       F  2001-12-07     Malaysia  Analytics   86.0  Course101
      4   3        Ng       John       M  2002-05-29    Singapore   Business   35.0  Course101
      5   3        Ng       John       M  2002-05-29    Singapore   Business   42.0  Course102
      6   4     Leong    Michael       F  2003-01-18    Singapore  Analytics   60.0  Course101
      7   4     Leong    Michael       F  2003-01-18    Singapore  Analytics   74.0  Course102
      8   5       Tay      Susan       M  2002-07-12    Singapore  Analytics   85.0  Course102
      9   5       Tay      Susan       M  2002-07-12    Singapore  Analytics   91.0  Course101
     10   6       Ong      Nancy       F  2001-11-30    Singapore    Finance   81.0  Course101
     11   6       Ong      Nancy       F  2001-11-30    Singapore    Finance   90.0  Course102
     12   7      Chan    Charlie       M  2000-06-22    Singapore  Analytics    NaN       None
     13   8  Kaminsky      Petra       F        None    Australia  Analytics    NaN  Course102
     14   8  Kaminsky      Petra       F        None    Australia  Analytics   62.0  Course101
     15   9       Lee     Thomas       M  2001-11-30    Indonesia  Analytics   79.0  Course101
     16   9       Lee     Thomas       M  2001-11-30    Indonesia  Analytics   88.0  Course102
     17  10       Tan      David       M  2002-10-04    Singapore    Finance    NaN  Course102
     18  10       Tan      David       M  2002-10-04    Singapore    Finance   54.0  Course101
```

Figure 6.40 Outer Join the Tables `students` and `grades` (1)



```
     19  11    Wikodo      Minna       F  2001-05-18     Malaysia    Analytics   59.0  Course102
     20  11    Wikodo      Minna       F  2001-05-18     Malaysia    Analytics   65.0  Course101
     21  12       Lim    Matthew       M  2002-10-01    Singapore    Analytics   63.0  Course102
     22  12       Lim    Matthew       M  2002-10-01    Singapore    Analytics   78.0  Course101
     23  13       Lee        Amy       F  2002-06-27    Singapore    Analytics   87.0  Course101
     24  13       Lee        Amy       F  2002-06-27    Singapore    Analytics   92.0  Course102
     25  14       Sim        Eva       F  2002-04-23    Singapore    Analytics   51.0  Course101
     26  14       Sim        Eva       F  2002-04-23    Singapore    Analytics   64.0  Course102
     27  15      Pang       Andy       M  2001-01-08     Malaysia  Accountancy   32.0  Course102
     28  15      Pang       Andy       M  2001-01-08     Malaysia  Accountancy   40.0  Course101
     29  16        He      Zhiyi       F  2002-05-07        China    Analytics   68.0  Course101
     30  16        He      Zhiyi       F  2002-05-07        China    Analytics   70.0  Course102
     31  17       Pho      Emily       F  2002-11-28    Singapore    Analytics   67.0  Course102
     32  17       Pho      Emily       F  2002-11-28    Singapore    Analytics   70.0  Course101
     33  18       Eng     Maggie       F  2001-09-16    Singapore    Analytics   55.0  Course102
     34  18       Eng     Maggie       F  2001-09-16    Singapore    Analytics   56.0  Course101
     35  19       Tan    Clement       M  2002-03-11    Singapore    Analytics   73.0  Course101
     36  19       Tan    Clement       M  2002-03-11    Singapore    Analytics   73.0  Course102
     37  20       Koh      Vicky       F  2002-07-31    Singapore    Analytics    NaN  Course102
     38  20       Koh      Vicky       F  2002-07-31    Singapore    Analytics   47.0  Course101
     39  21      None       None    None        None         None         None   79.0  Course102
     40  22      None       None    None        None         None         None   98.0  Course102
     41  23      None       None    None        None         None         None   51.0  Course102
     42  24      None       None    None        None         None         None   63.0  Course102
```

Figure 6.41 Outer Join the Tables `students` and `grades` (2)

From Figure 6.40, we can see that the student number 7, who has no exam records found in the table `grades`, is included in the output table. This is not surprising since it is the same result as the left join method. In Figure 6.41, the students with ID numbers 21 to 24 only seem to have some exam results but no personal records found in the table `students`. Basically, such cases can only be detected if we left join the table `grades` with the table `students`, which is exactly our second SELECT statement in Figure 6.40.

Read

Refer to the link below for more details and examples on the `UNION ALL` operator of SQL:

https://www.sqlitetutorial.net/sqlite-union/

Refer to the link below for more details and examples on the `.remove()` method:

https://www.w3schools.com/python/ref_list_remove.asp

Chapter 4: Grouping Data

Lesson Recording

Group Data with SQL

4.1 Combining Records into Groups

In SQL, we can combine records of a table into groups based on one or more categorical variables. In most cases, the grouping is combined with the calculation of some statistics for each group by the aggregate functions.

Table 6.3 List of Aggregate Functions in SQL

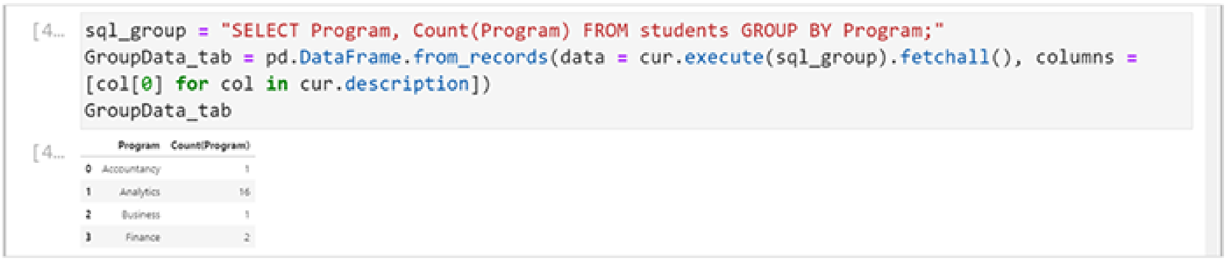
Aggregate Functions	Description
`AVG`	Average of the specified columns in a group
`COUNT`	Number of rows in a group
`MAX`	Maximum value of the specified columns in a group
`MIN`	Minimum value of the specified columns in a group
`STDDEV`	Standard deviation of the specified columns in a group
`SUM`	Sum of the specified columns in a group
`VARIANCE`	Variance of the specified columns in a group

To group records of a table together, we need to add the `GROUP BY` statement to the `SELECT` statement.

```
SELECT var_list AGGREGATE_FUNCTION(var_name)
FROM table_name
GROUP BY groupvar1_name, groupvar2_name, …;
```

In the variable list of the `SELECT` statement, we can also specify the aggregate function and the variable for which the aggregated statistics of each group should be calculated. The `GROUP BY` statement is followed by the variable names based on which the groups should be formed, and the variable names must be separated by commas here. If the table is grouped by more than one variables, the groups will be formed by the cartesian products of the categories in the variables.

Example (Cont'd): Suppose we would like to group the students by their study programmes and the number of students in each programme should be counted.



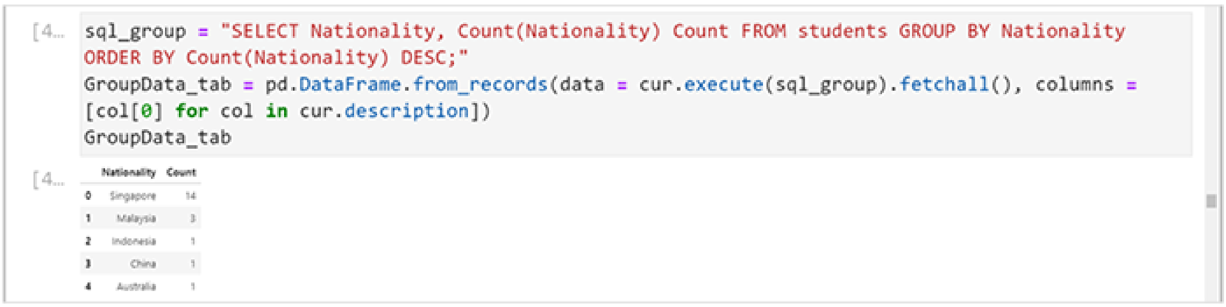
```
[4… sql_group = "SELECT Program, Count(Program) FROM students GROUP BY Program;"
    GroupData_tab = pd.DataFrame.from_records(data = cur.execute(sql_group).fetchall(), columns =
    [col[0] for col in cur.description])
    GroupData_tab
```

```
[4…        Program  Count(Program)
     0  Accountancy               1
     1    Analytics              16
     2     Business               1
     3      Finance               2
```

Figure 6.42 Count the Number of Students in Each Programme

In the above syntax, we only select the column `program` for our output tables since we are only interested in the number of students in each of its categories. The count of students is added as a variable to the query, and the name of it is simply taken over from the `SELECT` statement, namely `Count(Program)`.

We can also group and count our students by their nationalities and the result will be sorted by the counts in the descending order.

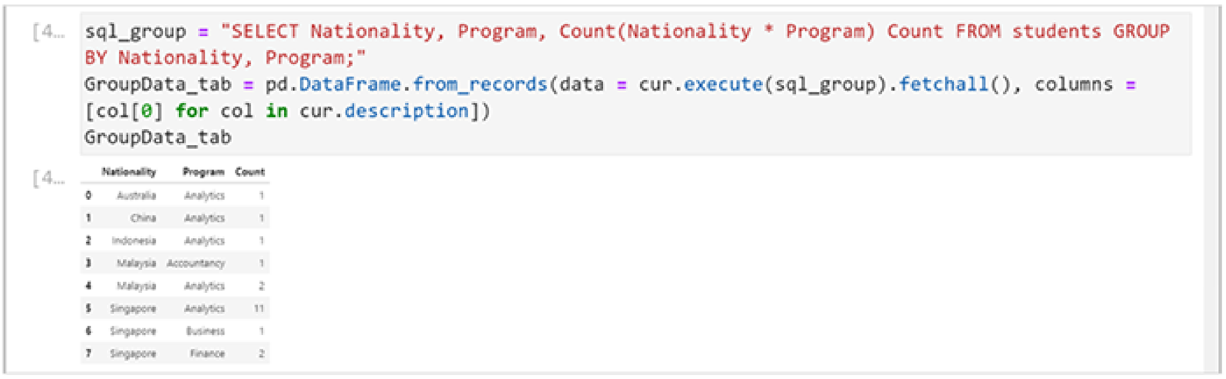


```
[4... sql_group = "SELECT Nationality, Count(Nationality) Count FROM students GROUP BY Nationality
     ORDER BY Count(Nationality) DESC;"
     GroupData_tab = pd.DataFrame.from_records(data = cur.execute(sql_group).fetchall(), columns =
     [col[0] for col in cur.description])
     GroupData_tab

[4...    Nationality  Count
      0    Singapore     14
      1     Malaysia      3
      2    Indonesia      1
      3        China      1
      4    Australia      1
```

Figure 6.43 Sort the Nationalities of the Students by Their Counts

In Figure 6.43, we add the name `Count` to the variable list in the `SELECT` statement which will be used as the variable name for `Count(Nationality)`. In other words, we can specify a name for the column of the aggregated statistics by placing it behind the aggregate function in the `SELECT` statement.

Next, we will group the students by their study programmes and nationalities. The frequency of each nationality in each study programme should be counted as well.



```
[4... sql_group = "SELECT Nationality, Program, Count(Nationality * Program) Count FROM students GROUP
     BY Nationality, Program;"
     GroupData_tab = pd.DataFrame.from_records(data = cur.execute(sql_group).fetchall(), columns =
     [col[0] for col in cur.description])
     GroupData_tab

[4...    Nationality      Program  Count
      0    Australia    Analytics      1
      1        China    Analytics      1
      2    Indonesia    Analytics      1
      3     Malaysia  Accountancy      1
      4     Malaysia    Analytics      2
      5    Singapore    Analytics     11
      6    Singapore     Business      1
      7    Singapore      Finance      2
```

Figure 6.44 Group the Students by their Study Programmes and Nationalities

The grouping variables, separated by a comma, are listed in the `GROUP BY` statement. To count the frequency of the cartesian product of `nationality` and `program`, we need to put a multiplication operator (`*`) between the two variables inside the `COUNT` function. From Figure 6.44, we can see that there are altogether eleven analytics students who are from Singapore, and two students of the same programme actually come from Malaysia, and so on.

Read

Refer to the links below for more details and examples on the `GROUP BY` operator of SQL:

https://www.w3schools.com/sql/sql_groupby.asp

https://www.sqlitetutorial.net/sqlite-group-by/

Refer to the link below for more details and examples on the `MIN()` and `MAX()` functions of SQL:

https://www.w3schools.com/sql/sql_min_max.asp

Refer to the link below for more details and examples on the `COUNT()`, `AVG()`, and `SUM()` functions of SQL:

https://www.w3schools.com/sql/sql_count_avg_sum.asp

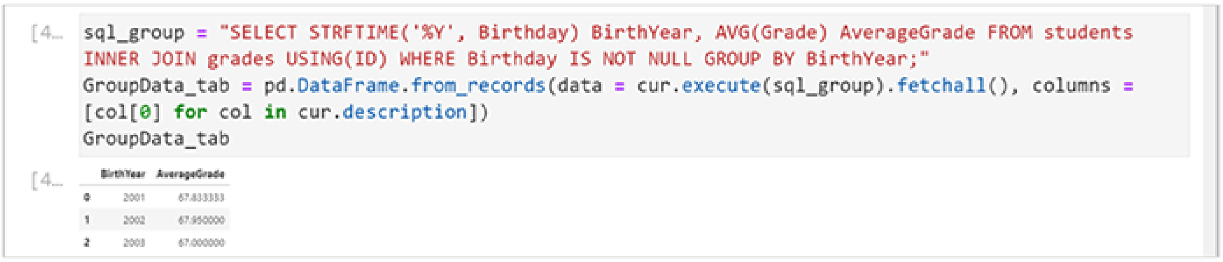
4.2 Filtering Groups

As demonstrated in the previous section, we can compute aggregated statistics after grouping the data. Furthermore, we can also filter the groups by some specified conditions. The filtering process for grouped data is carried out by the `HAVING` clause.

```
SELECT var_list AGGREGATE_FUNCTION(var_name)
FROM table_name
GROUP BY groupvar1_name, groupvar2_name, …
HAVING conditions;
```

Nevertheless, we can extend this `SELECT` statement with a `WHERE` clause that are used to filter the records before the grouping takes place, or with the `ORDER BY` keyword to sort the grouped table by the aggregated statistics, or the `INNER JOIN`/`LEFT JOIN`/`CROSS JOIN` clauses to merge columns from other tables before the grouping and the calculation of aggregated statistics are carried out.

Example (Cont'd): Suppose we would like to compare the average grades of the students based on their age. Instead of calculating their age, we can also just use their birthyears which can be extracted from the variable `Birthday` in the table `students` by the SQLite function `STRFTIME()`.



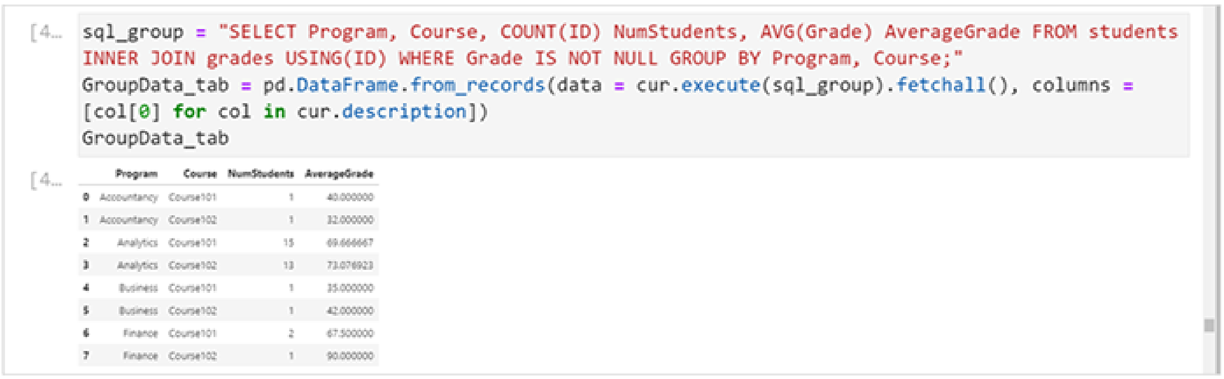
```
[4… sql_group = "SELECT STRFTIME('%Y', Birthday) BirthYear, AVG(Grade) AverageGrade FROM students
    INNER JOIN grades USING(ID) WHERE Birthday IS NOT NULL GROUP BY BirthYear;"
    GroupData_tab = pd.DataFrame.from_records(data = cur.execute(sql_group).fetchall(), columns =
    [col[0] for col in cur.description])
    GroupData_tab
```

```
[4…    BirthYear  AverageGrade
    0       2001     67.833333
    1       2002     67.950000
    2       2003     67.000000
```

Figure 6.45 Calculate Average Grade of Students from Different Birthyears

To retrieve the exam grades, we need to join the tables students and grades here. Since it does not make much sense to include the records with examination grades but no matching personal data as well as students who did not participate in any of the exams, inner join is applied. In the output table, we group the students by their birthyears generated by `STRFTIME('%Y', Birthday)`, which is included in the column list in the `SELECT` statement. The parameters in the function indicate that the year (`'%Y'`) should be extracted from the dates stored in the variable `Birthday`. We can also see from Figure 6.45 that once the column name of an aggregated statistic has been established within the `SELECT` statement, we can use it in the other parts of the syntax such as `WHERE`, `GROUP BY`, `ORDER BY`, etc.

In the next query, the students are grouped by their study programme and their average grades in each course in which they have taken the examination will be determined.



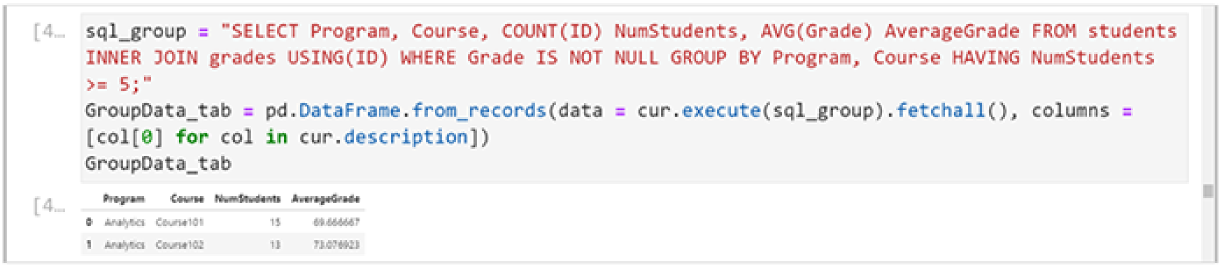
```
[4… sql_group = "SELECT Program, Course, COUNT(ID) NumStudents, AVG(Grade) AverageGrade FROM students
    INNER JOIN grades USING(ID) WHERE Grade IS NOT NULL GROUP BY Program, Course;"
    GroupData_tab = pd.DataFrame.from_records(data = cur.execute(sql_group).fetchall(), columns =
    [col[0] for col in cur.description])
    GroupData_tab
```

```
[4…      Program     Course  NumStudents  AverageGrade
     0  Accountancy  Course101           1     40.000000
     1  Accountancy  Course102           1     32.000000
     2    Analytics  Course101          15     69.666667
     3    Analytics  Course102          13     73.076923
     4     Business  Course101           1     35.000000
     5     Business  Course102           1     42.000000
     6      Finance  Course101           2     67.500000
     7      Finance  Course102           1     90.000000
```

Figure 6.46 Calculate Average Grade of Different Programmes in Different Courses

In the above program, we have excluded those records in the joined table that have missing values in the variable `Grade`. In other words, these were records from the table `students` for which matching could be found in the table `grades`, but their data in the column `Grade` were missing. We have also counted the number of different IDs in each of the groups to determine the number of students who participated in the corresponding examination. The counts can be found in the new column named `NumStudents`. As a result, there were 15 analytics students who took part in the examination of Course101 and their average grade is 69.67. Other programmes such as accountancy and business have only got one student each to participate in this examination. Therefore, their low average scores here are unreliable statistics for any inference.

If we would like to select only programmes with 5 students or more participated in the exams, we can use the `HAVING` clause for filtering the grouped table.

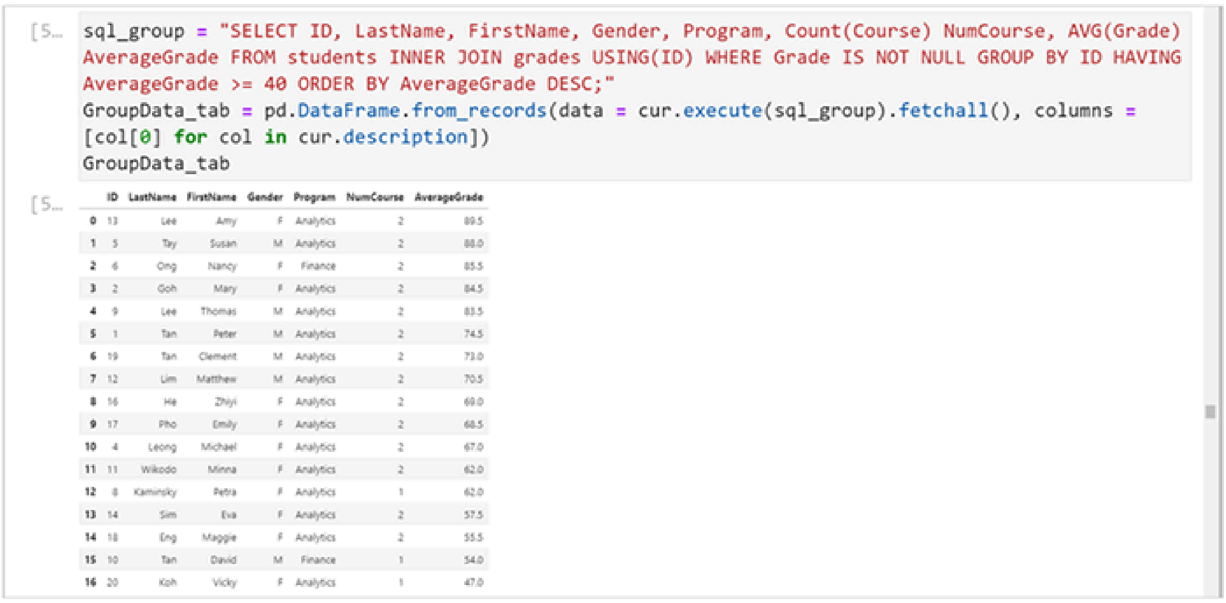


```
[4… sql_group = "SELECT Program, Course, COUNT(ID) NumStudents, AVG(Grade) AverageGrade FROM students
    INNER JOIN grades USING(ID) WHERE Grade IS NOT NULL GROUP BY Program, Course HAVING NumStudents
    >= 5;"
    GroupData_tab = pd.DataFrame.from_records(data = cur.execute(sql_group).fetchall(), columns =
    [col[0] for col in cur.description])
    GroupData_tab
```

```
[4…    Program     Course  NumStudents  AverageGrade
     0  Analytics  Course101          15     69.666667
     1  Analytics  Course102          13     73.076923
```

Figure 6.47 Selected Courses with `NumStudents` >= 5 Grouped by Study Programme

Now we will create a grouped table in which the average grade of each student in the two exams are calculated. The number of exam participations will be counted for each of them and stored as a new variable called `NumCourse`. Eventually, the output table should be sorted by the students' average grades in the descending order and contains only students with an average grade of at least 40 marks.



```
[5…  sql_group = "SELECT ID, LastName, FirstName, Gender, Program, Count(Course) NumCourse, AVG(Grade)
     AverageGrade FROM students INNER JOIN grades USING(ID) WHERE Grade IS NOT NULL GROUP BY ID HAVING
     AverageGrade >= 40 ORDER BY AverageGrade DESC;"
     GroupData_tab = pd.DataFrame.from_records(data = cur.execute(sql_group).fetchall(), columns =
     [col[0] for col in cur.description])
     GroupData_tab
```

[5…

	ID	LastName	FirstName	Gender	Program	NumCourse	AverageGrade
0	13	Lee	Amy	F	Analytics	2	89.5
1	5	Tay	Susan	M	Analytics	2	88.0
2	6	Ong	Nancy	F	Finance	2	85.5
3	2	Goh	Mary	F	Analytics	2	84.5
4	9	Lee	Thomas	M	Analytics	2	83.5
5	1	Tan	Peter	M	Analytics	2	74.5
6	19	Tan	Clement	M	Analytics	2	73.0
7	12	Lim	Matthew	M	Analytics	2	70.5
8	16	He	Zhiyi	F	Analytics	2	69.0
9	17	Pho	Emily	F	Analytics	2	68.5
10	4	Leong	Michael	F	Analytics	2	67.0
11	11	Wikodo	Minna	F	Analytics	2	62.0
12	8	Kaminsky	Petra	F	Analytics	1	62.0
13	14	Sim	Eva	F	Analytics	2	57.5
14	18	Eng	Maggie	F	Analytics	2	55.5
15	10	Tan	David	M	Finance	1	54.0
16	20	Koh	Vicky	F	Analytics	1	47.0

Figure 6.48 Listing Students with `AverageGrade >= 40`

In the first step, we drop variables that are less relevant to this query such as `Birthday`, `Nationality` and `Course`. Same as our code in Figure 6.47, records with missing data in the variable `Grade` will be dropped from the table. After grouping the table by the `ID` of the students and their average grades have been calculated, students will be sorted by their average grades and those with less than 40 marks will not be selected.

Read

Refer to the links below for more details and examples on the `HAVING` clause in SQL:

https://www.w3schools.com/sql/sql_having.asp

https://www.sqlitetutorial.net/sqlite-having/

Refer to the link below for more details and examples on the `STRFTIME()` function in SQLite:

https://sqlite.org/lang_datefunc.html

Refer to the link below for more details and examples on date data types in SQL:

https://www.w3schools.com/sql/sql_dates.asp

Chapter 5: Editing Data

Lesson Recording

Edit Data with SQL

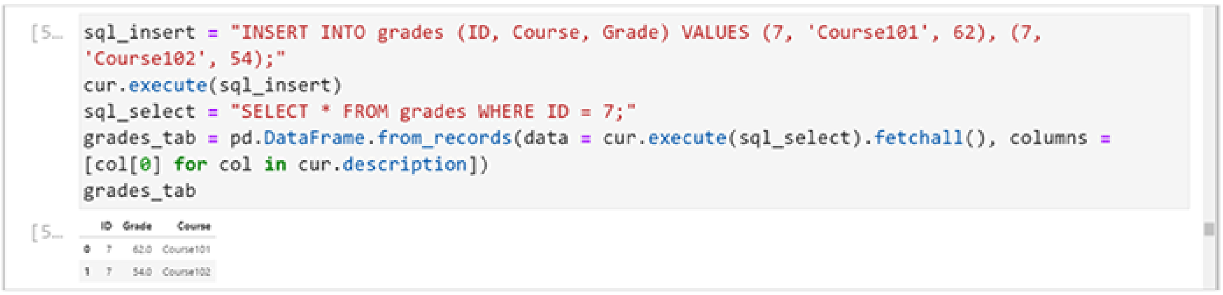
5.1 Inserting Records

In the previous chapters, we have been introduced to methods for extracting and reshaping information from a database. Nevertheless, SQL also provides the possibility for us to change the data or even the structure of a table. In this section, we will discuss how to insert new records to a table.

```
INSERT INTO table_name (var_list)
VALUES (value_list);
```

In the `INSERT INTO` statement, the variable list added behind the table name should be a subset of the column names in the table. Its length must be identical with the length of the value list, and both lists must be put in parentheses. It is also important to ensure that the sequence of the elements in the value list corresponds to the sequence of the variables so that the values are assigned to the correct column eventually. For inserting multiple records, the value list of each record must be wrapped up by a pair of brackets and every list must be separated by a comma from one another. Furthermore, the values of the variables excluded in the `INSERT INTO` statement for the new records will be `None`. If we intent to provide values to all columns, the variable list including the brackets can also be omitted from the syntax.

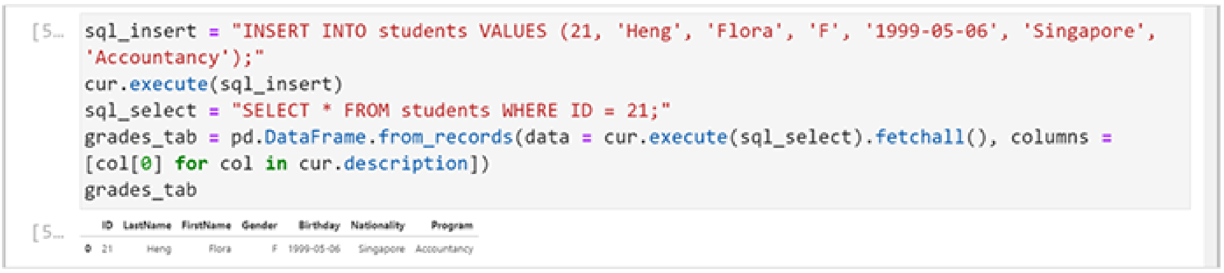
Example (Cont'd): In Chapter 4, we have often come across the student with ID number 7 who has no matching records found in the table `grades`. Suppose we have now received his grades for Course101 and Course102, which are 62 and 54, respectively, we can insert these two records into the table.



```
[5… sql_insert = "INSERT INTO grades (ID, Course, Grade) VALUES (7, 'Course101', 62), (7,
    'Course102', 54);"
    cur.execute(sql_insert)
    sql_select = "SELECT * FROM grades WHERE ID = 7;"
    grades_tab = pd.DataFrame.from_records(data = cur.execute(sql_select).fetchall(), columns =
    [col[0] for col in cur.description])
    grades_tab
[5…    ID  Grade     Course
    0   7   62.0  Course101
    1   7   54.0  Course102
```

Figure 6.49 Insert Multiple New Records for a Student into the Table `grades`

Recall from Figure 6.41 that the student with number 21 has got a grade in Course102 but no personal record in the table `students`. We have now received her personal data and would like to insert it into the table.



```
[5… sql_insert = "INSERT INTO students VALUES (21, 'Heng', 'Flora', 'F', '1999-05-06', 'Singapore',
    'Accountancy');"
    cur.execute(sql_insert)
    sql_select = "SELECT * FROM students WHERE ID = 21;"
    grades_tab = pd.DataFrame.from_records(data = cur.execute(sql_select).fetchall(), columns =
    [col[0] for col in cur.description])
    grades_tab
[5…    ID  LastName  FirstName  Gender     Birthday  Nationality      Program
    0  21      Heng      Flora       F   1999-05-06    Singapore  Accountancy
```

Figure 6.50 Insert a New Record into the Table `students` without Variable List

Different from the program in Figure 6.49, the variable list is omitted in the `INSERT INTO` statement here. Nevertheless, the behaviour of the program remains unchanged since the data of all variables are completely available and they are put in the right sequence in the syntax.

Read

Refer to the links below for more details and examples on the `INSERT INTO` clause in SQL:

https://www.w3schools.com/sql/sql_insert.asp

https://www.sqlitetutorial.net/sqlite-insert/

5.2 Updating Records

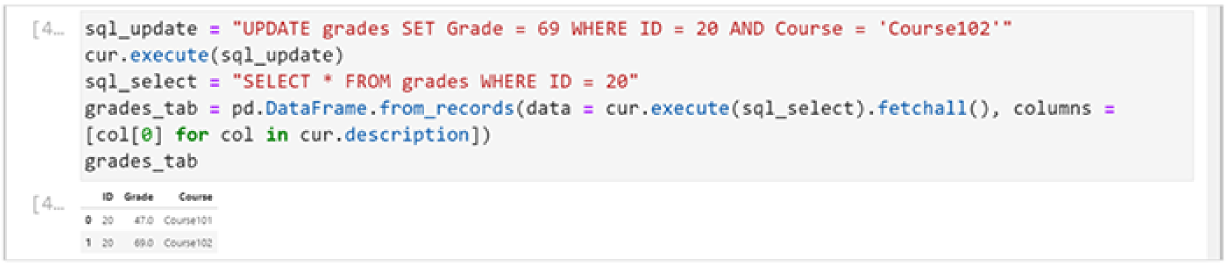
We can also update or edit the data of existing records by the `UPDATE` statement.

```
UPDATE table_name
SET var1_name = value1, var2_name = value2, …
WHERE condition;
```

The concept of updating data in the tables by SQL is slightly different from editing the contents of a spreadsheet. Here, we need to state certain conditions in a `WHERE` clause which must be fulfilled by a record in order to get itself updated. In other words, if the condition is true to more than one records, all of them will be modified simultaneously. Hence, depending on the nature of the update, the condition must be specified precisely so that the update is not applied to the wrong records. If the `WHERE` clause is omitted, *all* records in the involved table will be updated.

With the keyword `SET`, we can specify the columns that SQL should update and their new values. The `UPDATE` statement is particularly useful to replace missing values or outliers in a dataset.

Example (Cont'd): From Figure 6.41, we can see that student with ID number 20 has two records in the table `grades`. However, her grade in Course102 is missing. Suppose she had to take the supplementary exam due to illness and her grade is therefore only available with some weeks of delay.



```
[4… sql_update = "UPDATE grades SET Grade = 69 WHERE ID = 20 AND Course = 'Course102'"
    cur.execute(sql_update)
    sql_select = "SELECT * FROM grades WHERE ID = 20"
    grades_tab = pd.DataFrame.from_records(data = cur.execute(sql_select).fetchall(), columns =
    [col[0] for col in cur.description])
    grades_tab
```

```
[4…    ID  Grade     Course
    0  20   47.0  Course101
    1  20   69.0  Course102
```

Figure 6.51 Update the Value of a Selected Record in the Variable `Grade`

It requires two conditions here to find the target record for updating: the student ID number must be 20 and the course must be "Course102". If the `ID` number were not specified, every student's grade in Course102 would become 69. On the other hand, if `Course` were not included as one of the conditions, this student's grades of Course101 and Course102 would both change to 69.

Read

Refer to the links below for more details and examples on the `UPDATE` clause in SQL:

https://www.w3schools.com/sql/sql_update.asp

https://www.sqlitetutorial.net/sqlite-update/

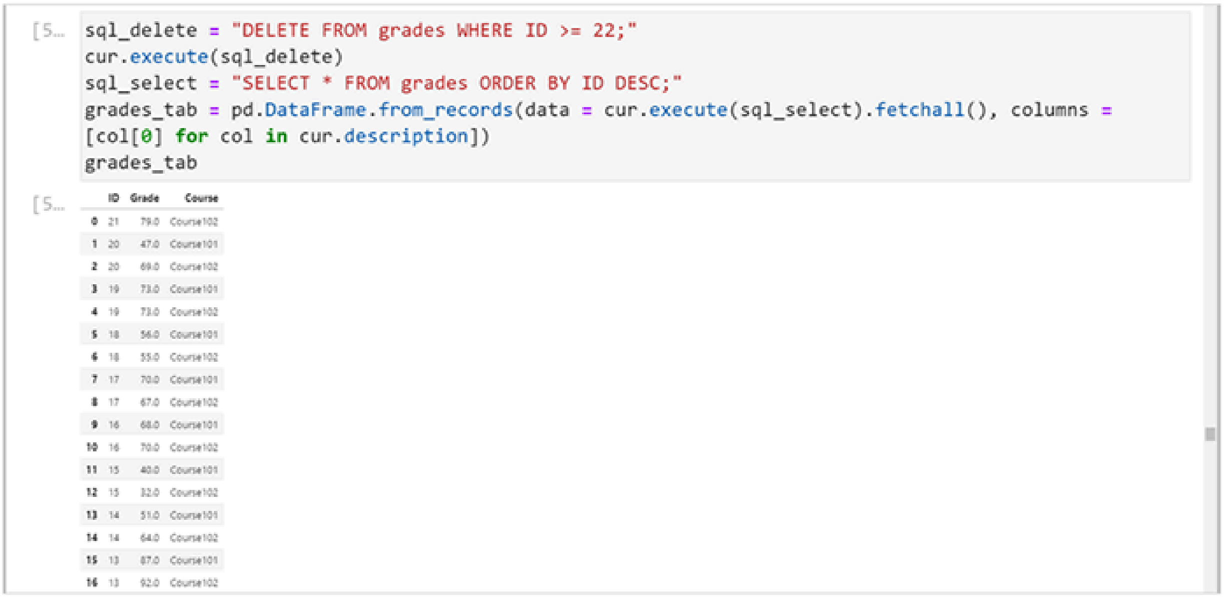
5.3 Deleting Records

Deleting records from a table works in a very similar way as updating data in rows. That is, conditions must be set so that records can be selected for removal.

```
DELETE FROM table_name
WHERE condition;
```

Same as the `UPDATE` statement, it is very important to specify the correct records for deletion. If the condition is too vague, there can be more records deleted than originally intended. Note that once a row has been dropped from a table, there is no possibility to undo it in SQL.

Example (Cont'd): After the personal data of student with ID number 21 has been inserted to the table `students` as shown in Figure 6.50, it is decided that all records of students with ID number 22 to 24 should be deleted from the table `grades` since no matching personal data could be found for them.



```
[5… sql_delete = "DELETE FROM grades WHERE ID >= 22;"
    cur.execute(sql_delete)
    sql_select = "SELECT * FROM grades ORDER BY ID DESC;"
    grades_tab = pd.DataFrame.from_records(data = cur.execute(sql_select).fetchall(), columns =
    [col[0] for col in cur.description])
    grades_tab
```

```
[5…     ID  Grade     Course
     0  21   79.0  Course102
     1  20   47.0  Course101
     2  20   69.0  Course102
     3  19   73.0  Course101
     4  19   73.0  Course102
     5  18   56.0  Course101
     6  18   55.0  Course102
     7  17   70.0  Course101
     8  17   67.0  Course102
     9  16   68.0  Course101
    10  16   70.0  Course102
    11  15   40.0  Course101
    12  15   32.0  Course102
    13  14   51.0  Course101
    14  14   64.0  Course102
    15  13   87.0  Course101
    16  13   92.0  Course102
```

Figure 6.52 Delete Records with ID Number >=22 from the Table `grades`

The condition for deletion is a rather simple one: `ID >= 22`. Since it was the records with the highest ID numbers being deleted, we can sort the table by `ID` in the descending order to check whether the deletion has been carried out properly.

Read

Refer to the links below for more details and examples on the `DELETE` clause in SQL:

https://www.w3schools.com/sql/sql_ref_delete.asp

https://www.sqlitetutorial.net/sqlite-delete/

5.4 Altering Tables

In the previous sections, we have discussed changing the rows of a table. In this section, we will introduce a method to alter a table by editing its columns. With the `ALTER TABLE` statement, we can rename a table, rename a column, or add a column. In the first step, we introduce the following syntax to add a column to a table.

```
ALTER TABLE table_name
ADD column_name;
```

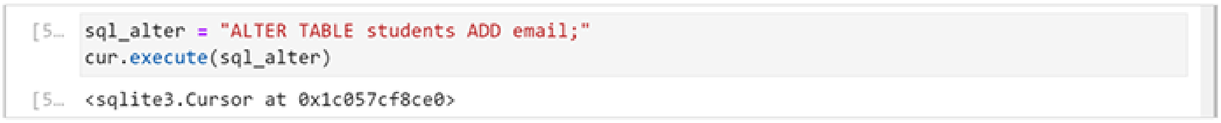
Unlike the other SQL versions, SQLite3 only allows adding one column at a time.

We can rename a column with the following version of the `ALTER TABLE` statement.

```
ALTER TABLE table_name
RENAME old_column_name TO new_column_name;
```

Here, we can only rename one column at a time as well.

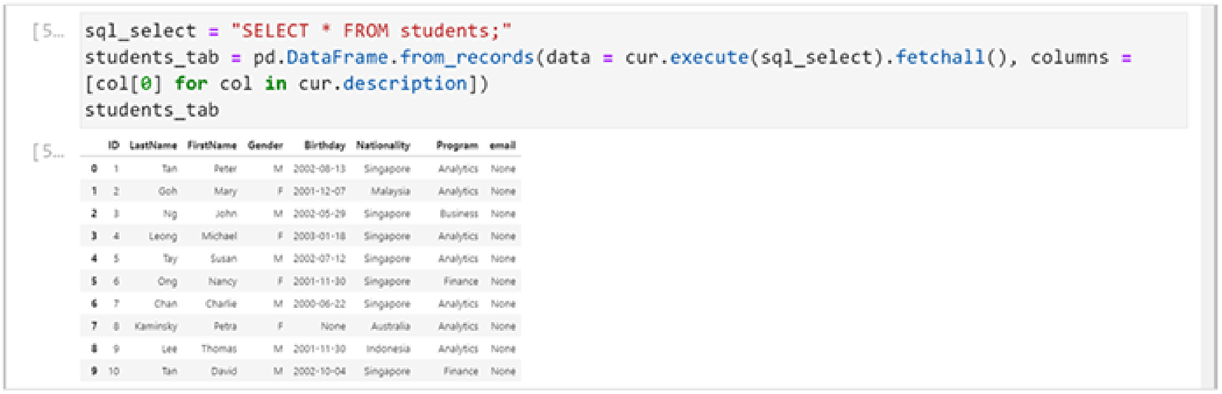
Example (Cont'd): We would now like to add a new column called `email` to the table `students` to store their email addresses.



```
[5… sql_alter = "ALTER TABLE students ADD email;"
    cur.execute(sql_alter)

[5… <sqlite3.Cursor at 0x1c057cf8ce0>
```

Figure 6.53 Add a New Column called `email` to the Table `students`

The variable is appended to the rightmost edge of the table with `None` as its value.

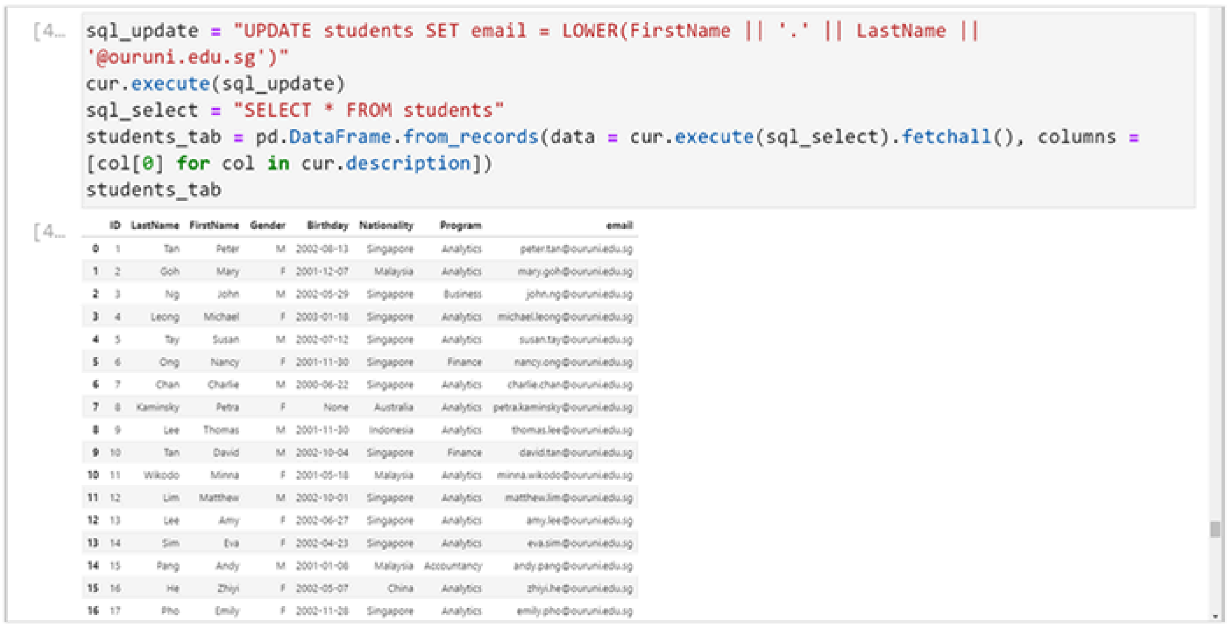


```
[5… sql_select = "SELECT * FROM students;"
    students_tab = pd.DataFrame.from_records(data = cur.execute(sql_select).fetchall(), columns =
    [col[0] for col in cur.description])
    students_tab
```

	ID	LastName	FirstName	Gender	Birthday	Nationality	Program	email
0	1	Tan	Peter	M	2002-08-13	Singapore	Analytics	None
1	2	Goh	Mary	F	2001-12-07	Malaysia	Analytics	None
2	3	Ng	John	M	2002-05-29	Singapore	Business	None
3	4	Leong	Michael	F	2003-01-18	Singapore	Analytics	None
4	5	Tay	Susan	M	2002-07-12	Singapore	Analytics	None
5	6	Ong	Nancy	F	2001-11-30	Singapore	Finance	None
6	7	Chan	Charlie	M	2000-06-22	Singapore	Analytics	None
7	8	Kaminsky	Petra	F	None	Australia	Analytics	None
8	9	Lee	Thomas	M	2001-11-30	Indonesia	Analytics	None
9	10	Tan	David	M	2002-10-04	Singapore	Finance	None

Figure 6.54 The Table `students` after a New Column `email` Being Added

Suppose the email address format of this university is "FirstName.LastName@ouruni.edu.sg". We can use the `||` operator of SQL to concatenate the values in the columns `FirstName` and `LastName` with the string `"@ouruni.edu.sg"` as the values of the new column `email`. Moreover, we can also convert all characters of the email address to lower case with the `LOWER()` function of SQL.



```
[4… sql_update = "UPDATE students SET email = LOWER(FirstName || '.' || LastName ||
    '@ouruni.edu.sg')"
    cur.execute(sql_update)
    sql_select = "SELECT * FROM students"
    students_tab = pd.DataFrame.from_records(data = cur.execute(sql_select).fetchall(), columns =
    [col[0] for col in cur.description])
    students_tab
```

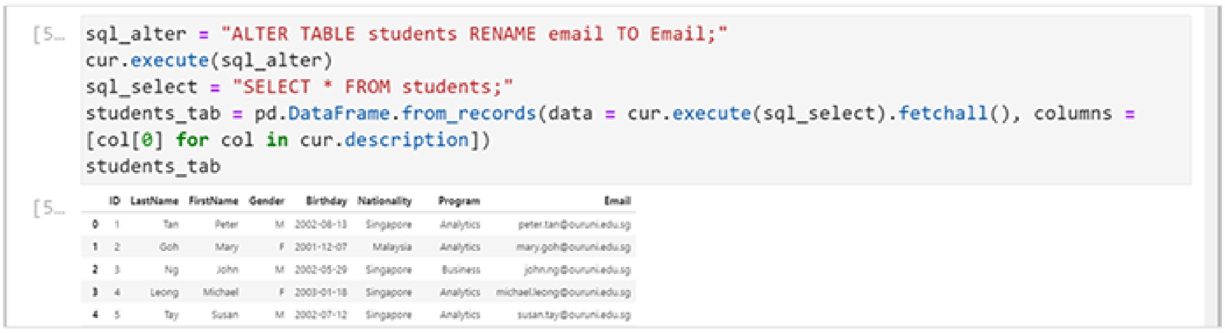
[4…

	ID	LastName	FirstName	Gender	Birthday	Nationality	Program	email
0	1	Tan	Peter	M	2002-08-13	Singapore	Analytics	peter.tan@ouruni.edu.sg
1	2	Goh	Mary	F	2001-12-07	Malaysia	Analytics	mary.goh@ouruni.edu.sg
2	3	Ng	John	M	2002-05-29	Singapore	Business	john.ng@ouruni.edu.sg
3	4	Leong	Michael	F	2003-01-18	Singapore	Analytics	michael.leong@ouruni.edu.sg
4	5	Tay	Susan	M	2002-07-12	Singapore	Analytics	susan.tay@ouruni.edu.sg
5	6	Ong	Nancy	F	2001-11-30	Singapore	Finance	nancy.ong@ouruni.edu.sg
6	7	Chan	Charlie	M	2000-06-22	Singapore	Analytics	charlie.chan@ouruni.edu.sg
7	8	Kaminsky	Petra	F	None	Australia	Analytics	petra.kaminsky@ouruni.edu.sg
8	9	Lee	Thomas	M	2001-11-30	Indonesia	Analytics	thomas.lee@ouruni.edu.sg
9	10	Tan	David	M	2002-10-04	Singapore	Finance	david.tan@ouruni.edu.sg
10	11	Wikodo	Minna	F	2001-05-18	Malaysia	Analytics	minna.wikodo@ouruni.edu.sg
11	12	Lim	Matthew	M	2002-10-01	Singapore	Analytics	matthew.lim@ouruni.edu.sg
12	13	Lee	Amy	F	2002-06-27	Singapore	Analytics	amy.lee@ouruni.edu.sg
13	14	Sim	Eva	F	2002-04-23	Singapore	Analytics	eva.sim@ouruni.edu.sg
14	15	Pang	Andy	M	2001-01-08	Malaysia	Accountancy	andy.pang@ouruni.edu.sg
15	16	He	Zhiyi	F	2002-05-07	China	Analytics	zhiyi.he@ouruni.edu.sg
16	17	Pho	Emily	F	2002-11-28	Singapore	Analytics	emily.pho@ouruni.edu.sg

Figure 6.55 Update Values in the Column `email`

The update here does not require the `WHERE` clause because all email addresses can be generated with the identical syntax. Certainly, the procedure here is strongly simplified since we do not consider the possibilities of a) students using user-defined email address, b) assigning the same email address to multiple students because their names are identical, c) white spaces in the students' name if they have middle names, d) special characters in their names such as å, é, ñ, or ß. We need a much more sophisticated program to deal with all these issues, and we will omit it since it is out of the scope of our discussion here.

Finally, since the names of all the other columns start with a capital letter, we will therefore rename the column from `email` to `Email`.



```
[5… sql_alter = "ALTER TABLE students RENAME email TO Email;"
    cur.execute(sql_alter)
    sql_select = "SELECT * FROM students;"
    students_tab = pd.DataFrame.from_records(data = cur.execute(sql_select).fetchall(), columns =
    [col[0] for col in cur.description])
    students_tab
```

```
[5…    ID  LastName  FirstName  Gender    Birthday  Nationality  Program                      Email
    0   1       Tan      Peter       M  2002-08-13    Singapore  Analytics    peter.tan@ouruni.edu.sg
    1   2       Goh       Mary       F  2001-12-07     Malaysia  Analytics     mary.goh@ouruni.edu.sg
    2   3        Ng       John       M  2002-05-29    Singapore   Business      john.ng@ouruni.edu.sg
    3   4     Leong    Michael       F  2003-01-18    Singapore  Analytics michael.leong@ouruni.edu.sg
    4   5       Tay      Susan       M  2002-07-12    Singapore  Analytics    susan.tay@ouruni.edu.sg
```

Figure 6.56 Rename the Column from `email` to `Email`

Beside altering the content of a table, we can also create a new table in the database.

```
CREATE TABLE table_name (column1_name, column2_name, …);
```

And it is also possible to drop a table from the database.

```
DROP TABLE table_name;
```

Certainly, we also have the possibility to rename a table.

```
ALTER TABLE table_name RENAME TO new_table_name;
```

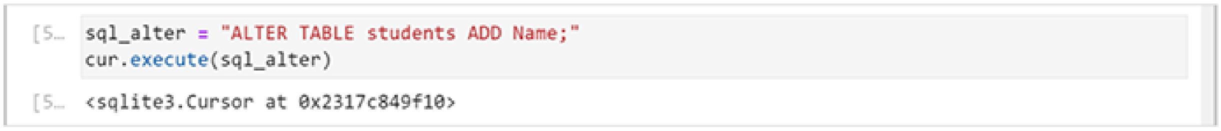
The syntax for renaming a table is fairly similar to the one for renaming a column in a table. The only difference is that there should be no name between `RENAME` and `TO` when giving a new name to a table.

If we want to copy the data from one table to another one which has the same column structure, we can modify and apply the `INSERT INTO` statement for this purpose.

```
INSERT INTO target_table_name
SELECT value_list
FROM source_table_name;
```

Instead of having a variable list and a value list, we can have the `SELECT` statement embedded in the above syntax. As a result, data are queried from another table first and then inserted into the target table. We can also use additional clauses such as `ORDER BY`, `WHERE`, `INNER JOIN`, etc., to sort or select specific records, or merge multiple tables before inserting them.

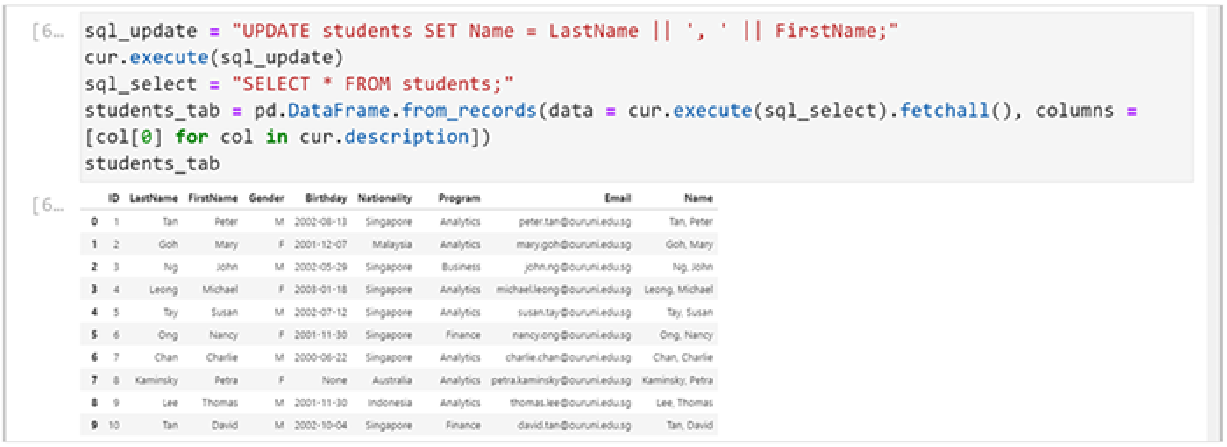
Example (Cont'd): In the following, we will first merge the last and the first names of the students to one new column called `Name` in the table `students`. Subsequently, we will duplicate the table and drop the variables `LastName` and `FirstName` from the new one and rearrange the sequence of the columns. The reasons we do not carry out the alteration in the existing table students is that we would like to keep the original record just in case we will need it again.



```
[5… sql_alter = "ALTER TABLE students ADD Name;"
    cur.execute(sql_alter)
[5… <sqlite3.Cursor at 0x2317c849f10>
```

Figure 6.57 Add a New Column Called `Name` to the Table `students`

In the next step, we convert the students' last name and first name to the format `"LastName, FirstName"` and store it in a new variable called `Name`.



```
[6… sql_update = "UPDATE students SET Name = LastName || ', ' || FirstName;"
    cur.execute(sql_update)
    sql_select = "SELECT * FROM students;"
    students_tab = pd.DataFrame.from_records(data = cur.execute(sql_select).fetchall(), columns =
    [col[0] for col in cur.description])
    students_tab
```

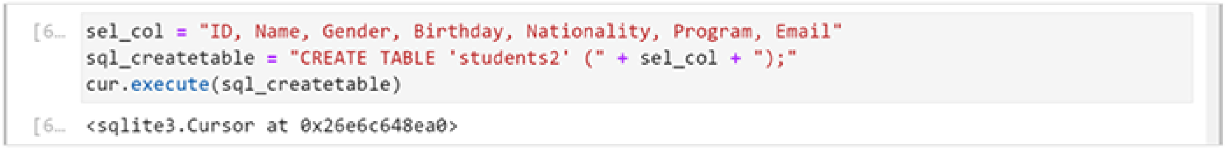
[6…

	ID	LastName	FirstName	Gender	Birthday	Nationality	Program	Email	Name
0	1	Tan	Peter	M	2002-08-13	Singapore	Analytics	peter.tan@ouruni.edu.sg	Tan, Peter
1	2	Goh	Mary	F	2001-12-07	Malaysia	Analytics	mary.goh@ouruni.edu.sg	Goh, Mary
2	3	Ng	John	M	2002-05-29	Singapore	Business	john.ng@ouruni.edu.sg	Ng, John
3	4	Leong	Michael	F	2003-01-18	Singapore	Analytics	michael.leong@ouruni.edu.sg	Leong, Michael
4	5	Tay	Susan	M	2002-07-12	Singapore	Analytics	susan.tay@ouruni.edu.sg	Tay, Susan
5	6	Ong	Nancy	F	2001-11-30	Singapore	Finance	nancy.ong@ouruni.edu.sg	Ong, Nancy
6	7	Chan	Charlie	M	2000-06-22	Singapore	Analytics	charlie.chan@ouruni.edu.sg	Chan, Charlie
7	8	Kaminsky	Petra	F	None	Australia	Analytics	petra.kaminsky@ouruni.edu.sg	Kaminsky, Petra
8	9	Lee	Thomas	M	2001-11-30	Indonesia	Analytics	thomas.lee@ouruni.edu.sg	Lee, Thomas
9	10	Tan	David	M	2002-10-04	Singapore	Finance	david.tan@ouruni.edu.sg	Tan, David

Figure 6.58 Concatenate Last Name and First Name as a New Variable

Same as the creation of the email address in Figure 6.55, we use the `||` operator to merge the last name, the separating comma and the first name together.

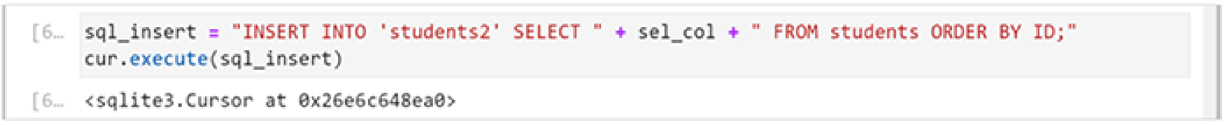
We can now copy the data to a new table called `students2`. However, we would like to exclude the columns `LastName` and `FirstName` in the new table since they are now completely represented by `Name`. Furthermore, as we can see from Figure 6.58, all new variables are appended to the rightmost end of the table. Thus, we need to rearrange the columns to our need first before copying them to the new table.



```
[6… sel_col = "ID, Name, Gender, Birthday, Nationality, Program, Email"
    sql_createtable = "CREATE TABLE 'students2' (" + sel_col + ");"
    cur.execute(sql_createtable)

[6… <sqlite3.Cursor at 0x26e6c648ea0>
```

Figure 6.59 Create a New Table named `students2`

In the first line, we generate a list of variables in the sequence of how they should be inserted in `students2`. The reason of using a Python string variable to store them here is that we will need to use it later for the `INSERT INTO` statement again. In the second line, we can then create the new table `students2` and use the variable list string `sel_col` to define the variables and their sequence in the new table.

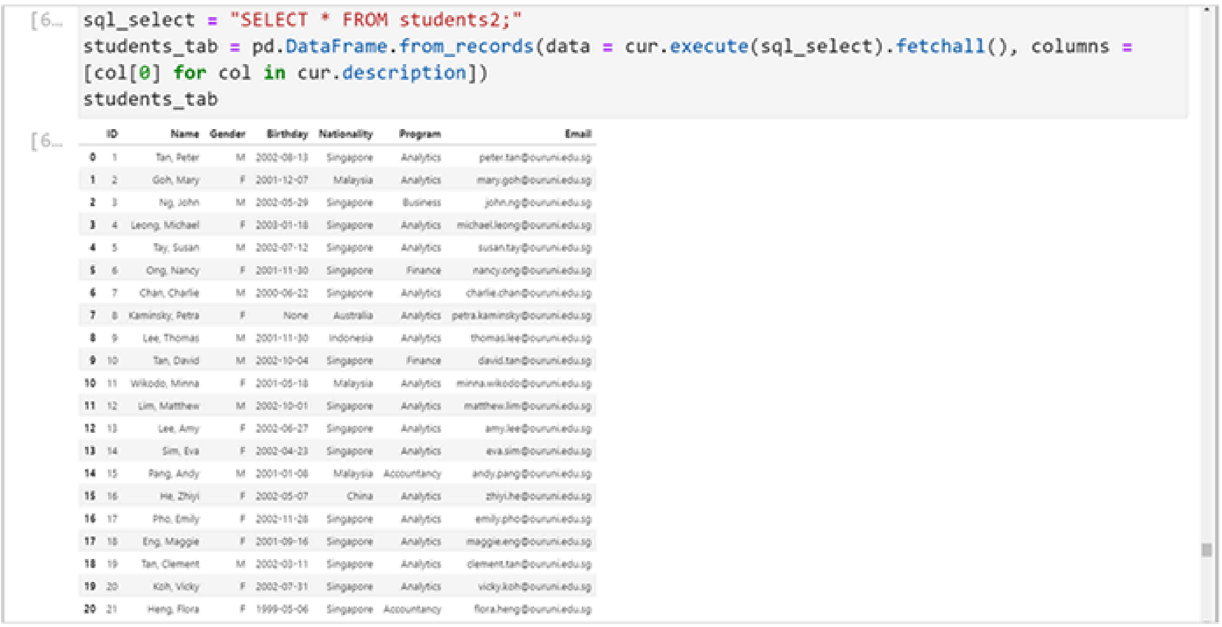


```
[6… sql_insert = "INSERT INTO 'students2' SELECT " + sel_col + " FROM students ORDER BY ID;"
    cur.execute(sql_insert)

[6… <sqlite3.Cursor at 0x26e6c648ea0>
```

Figure 6.60 Query Data from `students` and Insert Them Into `students2`

In Figure 6.60, we combine the `SELECT` and the `INSERT INTO` statements to transfer the data from `students` to `students2`. We insert the variable list string `sel_cols`, which has already been used in the `CREATE TABLE` statement in Figure 6.59, into the `SELECT` statement. Hence, we have ensured that the sequence and names of the columns are identical in both the programs. In addition, we added the `ORDER BY` keyword to the `SELECT` statement so that the data are now sorted by the ID numbers of the students.

Figure 6.61 shows the table `students2` after the data have been inserted.



```
[6… sql_select = "SELECT * FROM students2;"
    students_tab = pd.DataFrame.from_records(data = cur.execute(sql_select).fetchall(), columns =
    [col[0] for col in cur.description])
    students_tab
```

[6…

	ID	Name	Gender	Birthday	Nationality	Program	Email
0	1	Tan, Peter	M	2002-08-13	Singapore	Analytics	peter.tan@oururi.edu.sg
1	2	Goh, Mary	F	2001-12-07	Malaysia	Analytics	mary.goh@oururi.edu.sg
2	3	Ng, John	M	2002-05-29	Singapore	Business	john.ng@oururi.edu.sg
3	4	Leong, Michael	F	2003-01-18	Singapore	Analytics	michael.leong@oururi.edu.sg
4	5	Tay, Susan	M	2002-07-12	Singapore	Analytics	susan.tay@oururi.edu.sg
5	6	Ong, Nancy	F	2001-11-30	Singapore	Finance	nancy.ong@oururi.edu.sg
6	7	Chan, Charlie	M	2000-06-22	Singapore	Analytics	charlie.chan@oururi.edu.sg
7	8	Kaminsky, Petra	F	None	Australia	Analytics	petra.kaminsky@oururi.edu.sg
8	9	Lee, Thomas	M	2001-11-30	Indonesia	Analytics	thomas.lee@oururi.edu.sg
9	10	Tan, David	M	2002-10-04	Singapore	Finance	david.tan@oururi.edu.sg
10	11	Wikodo, Minna	F	2001-05-18	Malaysia	Analytics	minna.wikodo@oururi.edu.sg
11	12	Lim, Matthew	M	2002-10-01	Singapore	Analytics	matthew.lim@oururi.edu.sg
12	13	Lee, Amy	F	2002-06-27	Singapore	Analytics	amy.lee@oururi.edu.sg
13	14	Sim, Eva	F	2002-04-23	Singapore	Analytics	eva.sim@oururi.edu.sg
14	15	Pang, Andy	M	2001-01-08	Malaysia	Accountancy	andy.pang@oururi.edu.sg
15	16	He, Zhiyi	F	2002-05-07	China	Analytics	zhiyi.he@oururi.edu.sg
16	17	Pho, Emily	F	2002-11-28	Singapore	Analytics	emily.pho@oururi.edu.sg
17	18	Eng, Maggie	F	2001-09-16	Singapore	Analytics	maggie.eng@oururi.edu.sg
18	19	Tan, Clement	M	2002-03-11	Singapore	Analytics	clement.tan@oururi.edu.sg
19	20	Koh, Vicky	F	2002-07-31	Singapore	Analytics	vicky.koh@oururi.edu.sg
20	21	Heng, Flora	F	1999-05-06	Singapore	Accountancy	flora.heng@oururi.edu.sg

Figure 6.61 Output of students2 after Inserting the Data

Unlike other SQL versions, SQLite does not support `DROP COLUMN` in the `ALTER TABLE` statement. Therefore, the method introduced in Figure 6.58 and Figure 6.59, namely, to transfer all data except the variables that should be removed to a new table, is the only way to delete a column.

Read

Refer to the links below for more details and examples on the `ALTER TABLE` statement in SQL:

https://www.w3schools.com/sql/sql_alter.asp

https://www.sqlitetutorial.net/sqlite-alter-table/

Refer to the link below for more details and examples on the `RENAME TO` keywords of the `ALTER TABLE` statement in SQL:

https://www.sqlitetutorial.net/sqlite-rename-column/

Refer to the link below for more details and examples on the `LOWER()` function in SQL:

https://www.w3resource.com/sql/character-functions/lower.php

Refer to the links below for more details and examples on the `CREATE TABLE` statement in SQL:

https://www.w3schools.com/sql/sql_create_table.asp

https://www.sqlitetutorial.net/sqlite-create-table/

Refer to the links below for more details and examples on the `DROP TABLE` statement in SQL:

https://www.w3schools.com/sql/sql_drop_table.asp

https://www.sqlitetutorial.net/sqlite-drop-table/

5.5 Committing Changes in Database

It is often important to check on the existing content in the database and clear up tables that are no longer necessary. In SQLite, there is a master table called `sqlite_master` which holds the schema of the entire database. We can therefore query the names of the existing tables in a database by a `SELECT` statement.

```
SELECT name FROM sqlite_master WHERE type = 'table';
```

This syntax is a fixed expression and needs no adjustments. Basically, it queries the names of all the existing tables (obviously stored as values in the column `name`) from `sqlite_master` in the same fashion as we query data from a table. Armed with the result of this query, we can decide on whether taking actions or not on the tables.

So far, all procedures have actually been carried out on the virtual platform. That means, the changes are only stored in the virtual memory of our computer and not saved to the hard disk yet. Therefore, before closing the database, we need to commit all changes through the connection object back to the physical file of our database.

```
connection_object.commit()
```

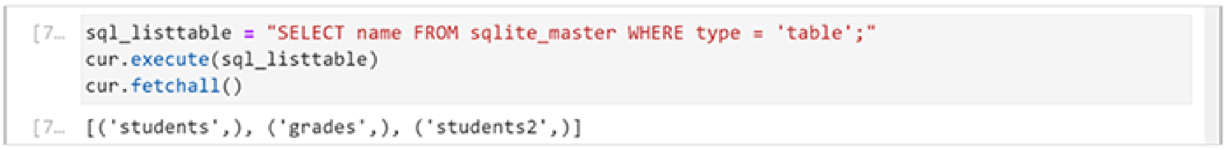
The `.commit()` method of the sqlite3 package must be applied to the connection object. It works in the same way as the "save" function in most of the software. That means, we can place it in our Python program wherever we think we need to save the changes before they are lost. However, we cannot undo the changes once they are committed.

Finally, we can close the connection to the database by the `.close()` method.

```
connection_object.close()
```

Note that the `.close()` method does not call the `.commit()` method automatically. In other words, if we close the connection before committing the changes to the physical file, all the modifications in the database will be lost.

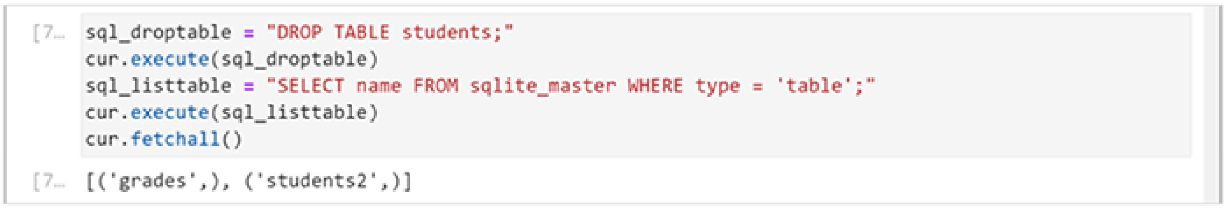
Example (Cont'd): In the first step, we would like to obtain a list of all the existing tables in the database.



```
[7… sql_listtable = "SELECT name FROM sqlite_master WHERE type = 'table';"
    cur.execute(sql_listtable)
    cur.fetchall()

[7… [('students',), ('grades',), ('students2',)]
```

Figure 6.62 Check on Existing Tables in the Database

As expected and confirmed by Figure 6.62, `students`, `grades` and `students2` are the tables that our database contains. Suppose we have now decided not to keep the table `students` anymore since `students2` actually contains the same data, we can apply the `DROP TABLE` statement to remove it.

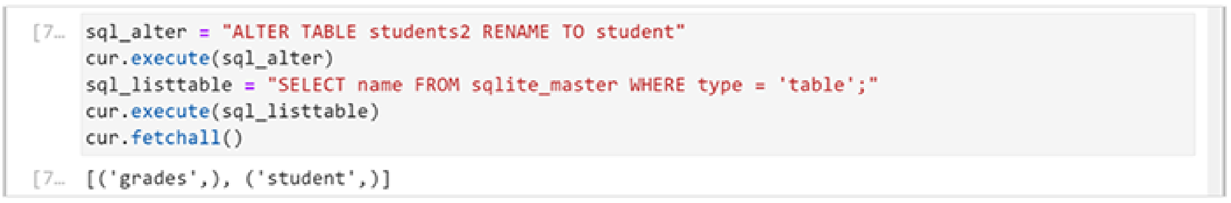


```
[7… sql_droptable = "DROP TABLE students;"
    cur.execute(sql_droptable)
    sql_listtable = "SELECT name FROM sqlite_master WHERE type = 'table';"
    cur.execute(sql_listtable)
    cur.fetchall()

[7… [('grades',), ('students2',)]
```

Figure 6.63 Dropping Table `students` from the Database

In Figure 6.63, we have added the `SELECT` statement to check on the names of the existing tables in the database after dropping `students`. The output of the query shows us that the table `students` has indeed disappeared.

Since `students` no longer exists, we can use `students2` to replace `student` by renaming it accordingly.

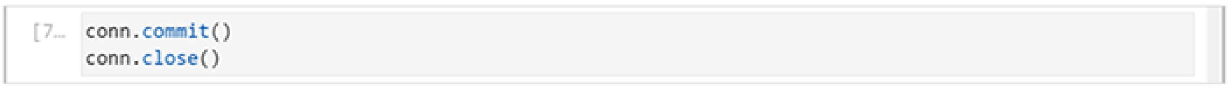


```
[7… sql_alter = "ALTER TABLE students2 RENAME TO student"
    cur.execute(sql_alter)
    sql_listtable = "SELECT name FROM sqlite_master WHERE type = 'table';"
    cur.execute(sql_listtable)
    cur.fetchall()
[7… [('grades',), ('student',)]
```

Figure 6.64 Renaming `students2` to `student`

By adding the same `SELECT` statement to check on the names of the existing tables, we can see that `students2` has indeed been renamed to `student`.

We can now use the object `conn`, which connects our database to the physical file "StudentsDB.db" according to Figure 6.5, to commit and save all the changes in our database. Subsequently, we can also close the connection to the database.



```
[7… conn.commit()
    conn.close()
```

Figure 6.65 Commit Changes of the Database to Physical File and Close Connection

Read

Refer to the link below for more details and examples on the `.commit()` method in the sqlite3 package:

https://docs.python.org/3/library/sqlite3.html#connection-objects

Summary

In this study unit, we have first learned how to write Python programs to store data entered by a user to a .csv text file. After the sqlite3 package has been introduced, we are able to connect Python with the databases and convert external data sources saved as .csv text files to database tables by SQL. With the `SELECT` statement, we could execute different types of data query such as sorting and filtering data. Here, we have used Python programming to generate `SELECT` statements flexibly and to convert the query output to pandas DataFrames for better presentation in the Python environment. We have also come across the four methods for joining two or more tables of a database: inner join, left join, cross join, and outer join. The output of these methods could vary strongly since they select the records differently. The option of grouping the data in a table and calculating some aggregated statistics has also been illustrated. And we have learned that groups can be filtered by the aggregated results. With the `ALTER TABLE` statements, we can also add, rename, or delete columns and alter thereby the structure of a table. Finally, we need to apply the `.commit()` method of the sqlite3 package to save the modification of the database to a physical file in our computer.

Formative Assessment

1. Which of the following modes of the `open()` function does not allow you to write to the source file?
   a. `"a"`
   b. `"r"`
   c. `"r+"`
   d. `"w"`

2. What role does the cursor object play in Python?
   a. It carries our commands from Python to SQL.
   b. It connects Python with the database.
   c. It is the database object in Python.
   d. It defines the table in the database with direct access.

3. After querying a SQL table with 50 observations, we use `fetchone()` to check on the outcome of one record for merely one time. How many records will remain available in the query output after that?
   a. 0
   b. 1
   c. 49
   d. 50

4. Which statement/clause/keyword cannot be embedded in the `SELECT` statement?
   a. `CROSS JOIN`
   b. `GROUP BY`
   c. `ORDER BY`
   d. `RENAME TO`

5. Table A has 50 records and table B has 70. A total of 40 records could be matched based on certain conditions. What would be the number of records in the output table if we merged A and B by inner join and cross join, respectively?
   a. 40 and 3500
   b. 50 and 3500
   c. 40 and 50
   d. 50 and 70

6. Which of the following values would not be selected given the following SQL command?

```
SELECT * FROM city_list WHERE city LIKE 'S%g%';
```

   a. Singapore.
   b. SINGAP.
   c. S'PORE
   d. SG

7. Which of the following is one of the main differences between `USING` and `ON` in an `INNER JOIN` clause?
   a. Only `ON` is allowed to use in an `INNER JOIN` clause.
   b. The matching variable from the second table will not be carried over to the output table of the query.
   c. Missing values of the matching variable from the first table will be replaced by the values of the matching variable from the second table.
   d. We can omit the names of the original tables when using `ON` in the `INNER JOIN` clause.

8. In a table called `cars`, there is a variable named type which has 4 values: A, B, C and D. Type A has 20 records, Type B has 10, Type C has 34, Type D has 7 records. What would be the output table of the following query?

```
SELECT type, COUNT(type)
FROM cars
GROUP BY type
WHERE COUNT(type) >= 20;
```

   a. Empty table
   b. Error
   c. Type C, 34
   d. Type A, 20
      Type C, 34

9. Which of the following functions is directly supported by SQLite or SQLite3?
   a. `OUTER JOIN`
   b. `DROP COLUMN`
   c. `REARRANGE COLUMN`
   d. `RENAME TABLE TO`

10. Which of the following is no more possible after the connection to a database is closed?
   a. Check the name of the existing tables in a database
   b. Connect to another database
   c. Rename the database file using file explorer of the operating system
   d. Extract information from a Python object which contains the content of an SQL query

Solutions or Suggested Answers

Formative Assessment

1. Which of the following modes of the `open()` function does not allow you to write to the source file?

   a. `"a"`

      Incorrect. We can append new records to the file in the appending mode.

   b. `"r"`

      Correct. We can only extract data from the file in the reading mode.

   c. `"r+"`

      Incorrect. We can add new contents to the file in the updating mode.

   d. `"w"`

      Incorrect. We can write to the file in the writing mode.

2. What role does the cursor object play in Python?

   a. It carries our commands from Python to SQL.

      Correct. We send our SQL commands through the cursor object from Python to SQL.

   b. It connects Python with the database.

      Incorrect. The connection object connects Python to the database.

   c. It is the database object in Python.

      Incorrect. There is no direct database object in Python.

   d. It defines the table in the database with direct access.

Incorrect. The cursor object does not specify the table in the database that we are working on.

3. After querying a SQL table with 50 observations, we use `fetchone()` to check on the outcome of one record for merely one time. How many records will remain available in the query output after that?

   a. 0

   Incorrect. The `fetchone()` function only fetches one record from the query output. So, there must be more than 0 records remaining.

   b. 1

   Incorrect. The `fetchone()` function only fetches one record from the query output. So, there must be more than 1 records remaining.

   c. 49

   Correct. Since the `fetchone()` function only fetches one record from the query output, there must be 49 records remaining.

   d. 50

   Incorrect. The `fetchone()` function fetches one record from the query output anyway. So, there must be less than 50 records remaining.

4. Which statement/clause/keyword cannot be embedded in the `SELECT` statement?

   a. `CROSS JOIN`

   Incorrect. The `CROSS JOIN` clause must be embedded in the `SELECT` statement.

   b. `GROUP BY`

Incorrect. The `GROUP BY` statement must be embedded in the `SELECT` statement.

c. `ORDER BY`

Incorrect. The `ORDER BY` keyword must be embedded in the `SELECT` statement.

d. `RENAME TO`

Correct. The `RENAME TO` keyword must be embedded in the `ALTER TABLE` statement.

5. Table A has 50 records and table B has 70. A total of 40 records could be matched based on certain conditions. What would be the number of records in the output table if we merged A and B by inner join and cross join, respectively?

a. 40 and 3500

Correct. Inner join creates the intersection set of both tables, i.e., 40, and cross join returns the cartesian product of both tables, i.e., 3500.

b. 50 and 3500

Incorrect. Only if A were left joined by B, the number of records of A would return, i.e., 50, and cross join returns the cartesian product of both tables, i.e., 3500.

c. 40 and 50

Incorrect. Inner join creates the intersection set of both tables, i.e., 40, but only if A were left joined by B, the number of records of A would return, i.e., 50.

d. 50 and 70

Incorrect. Only if A were left joined by B, the number of records of A would return, i.e., 50, and only if B were left joined by A, the number of records of B would return, i.e., 70.

6. Which of the following values would not be selected given the following SQL command?

```
SELECT * FROM city_list WHERE city LIKE 'S%g%';
```

   a. Singapore.

      Incorrect. Since “S” and “g” are parts of this string, SQL would select this value.

   b. SINGAP.

      Incorrect. Since “S” and “g” are parts of this string and SQL is not case sensitive, SQL would select this value.

   c. S’PORE

      Correct. Since “g” is not a sub-string of this string, SQL would not select this value.

   d. SG

      Incorrect. Since “S” and “g” are parts of this string and SQL is not case sensitive, SQL would select this value.

7. Which of the following is one of the main differences between `USING` and `ON` in an `INNER JOIN` clause?

   a. Only `ON` is allowed to use in an `INNER JOIN` clause.

      Incorrect. `USING` and `ON` are allowed to use in all `JOIN` clauses.

b. The matching variable from the second table will not be carried over to the output table of the query.

Correct. Only the matching variable from the first table will be carried over to the output table when `USING` is used in the `JOIN` clauses.

c. Missing values of the matching variable from the first table will be replaced by the values of the matching variable from the second table.

Incorrect. It is simply impossible to find matches for missing values in the matching variables from both tables. As a result, there cannot be any replacement.

d. We can omit the names of the original tables when using `ON` in the `INNER JOIN` clause.

Incorrect. We must include the names of the original tables when using `ON` in the `INNER JOIN` clause.

8. In a table called `cars`, there is a variable named type which has 4 values: A, B, C and D. Type A has 20 records, Type B has 10, Type C has 34, Type D has 7 records. What would be the output table of the following query?

```
SELECT type, COUNT(type)
FROM cars
GROUP BY type
WHERE COUNT(type) >= 20;
```

a. Empty table

Incorrect. The result would only be possible if the last line were `HAVING COUNT(type) >= 40`.

b. Error

Correct. We cannot select groups using the `WHERE` clause on the aggregated results. SQL will return an error to us.

c. Type C, 34

Incorrect. The result would only be possible if the last line were `HAVING COUNT(type) >= 30`.

d. Type A, 20
Type C, 34

Incorrect. The result would only be possible if the last line were `HAVING COUNT(type) >= 20`.

9. Which of the following functions is directly supported by SQLite or SQLite3?

a. `OUTER JOIN`

Incorrect. `OUTER JOIN` is not supported by SQLite. We can only outer join two tables using two `LEFT JOIN` clauses and connecting them with the `UNION ALL` operator.

b. `DROP COLUMN`

Incorrect. `DROP COLUMN` is not supported by SQLite. We can only drop a column from a table by creating a new table first, defining all the columns except the one that should be dropped and transferring the corresponding data to the new table.

c. `REARRANGE COLUMN`

Incorrect. `REARRANGE COLUMN` is not an SQL function at all. We can rearrange the columns from a table by creating a new table first, defining all the columns in the new sequence and transferring the corresponding data to the new table.

d. `RENAME TABLE TO`

Correct. `RENAME TABLE TO` is supported by SQLite. But it is not a stand-alone statement. We can rename the table using the `ALTER TABLE RENAME TO` statement.

10. Which of the following is no more possible after the connection to a database is closed?

    a. Check the name of the existing tables in a database

    Correct. To check the name of the existing tables in a database, we need to send a query to the master table `sql_master`, which means that we need the connection to the database.

    b. Connect to another database

    Incorrect. Once the connection to one database is closed, we can connect Python to another database.

    c. Rename the database file using file explorer of the operating system

    Incorrect. If we rename the database file outside Python or SQL, we can do it after the connection is closed and the file is not opened by another program anymore.

    d. Extract information from a Python object which contains the content of an SQL query

    Incorrect. Since it is a Python object, we do not need a connection to the database to work on it.

References

pandas. (n.d.). *pandas.DataFrame.from_records*. The pandas development team. https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.from_records.html

pandas. (n.d.). *pandas.DataFrame.to_sql*. The pandas development team. https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.to_sql.html

Python.org. (n.d.). *Built-in functions*. Python Software Foundation. https://docs.python.org/3/library/functions.html#open

Python.org. (n.d.). *Built-in types*. Python Software Foundation. https://docs.python.org/3/library/stdtypes.html#str.join

Python.org. (n.d.). *Input and output*. Python Software Foundation. https://docs.python.org/3/tutorial/inputoutput.html#tut-files

Python.org. (n.d.). *sqlite3 — DB-API 2.0 interface for SQLite databases*. Python Software Foundation. https://docs.python.org/3/library/sqlite3.html

Python.org. (n.d.). *sqlite3 — DB-API 2.0 interface for SQLite databases*. Python Software Foundation. https://docs.python.org/3/library/sqlite3.html#connection-objects

Python.org. (n.d.). *sqlite3 — DB-API 2.0 interface for SQLite databases*. Python Software Foundation. https://docs.python.org/3/library/sqlite3.html#sqlite3.Cursor.description

Python Tutorial. (n.d.). *Python list remove() method*. w3schools.com. https://www.w3schools.com/python/ref_list_remove.asp

SQL Tutorial. (n.d.). *SQL ALTER keyword*. w3schools.com. https://www.w3schools.com/sql/sql_alter.asp

SQL Tutorial. (n.d.). *SQL AND, OR and NOT operators*. w3schools.com. https://www.w3schools.com/sql/sql_and_or.asp

SQL Tutorial. (w3schools.com). *SQL BETWEEN operator*. https://www.w3schools.com/sql/sql_between.asp

SQL Tutorial. (n.d.). *SQL COUNT(), AVG() and SUM() functions*. w3schools.com. https://www.w3schools.com/sql/sql_count_avg_sum.asp

SQL Tutorial. (n.d.). *SQL CREATE TABLE statement*. w3schools.com. https://www.w3schools.com/sql/sql_create_table.asp

SQL Tutorial. (n.d.). *SQL DELETE keyword*. w3schools.com. https://www.w3schools.com/sql/sql_ref_delete.asp

SQL Tutorial. (n.d.). *SQL DROP TABLE statement*. w3schools.com. https://www.w3schools.com/sql/sql_drop_table.asp

SQL Tutorial. (n.d.). *SQL GROUP BY statement*. w3schools.com. https://www.w3schools.com/sql/sql_groupby.asp

SQL Tutorial. (n.d.). *SQL HAVING clause*. w3schools.com. https://www.w3schools.com/sql/sql_having.asp

SQL Tutorial. (n.d.). *SQL IN operator*. w3schools.com. https://www.w3schools.com/sql/sql_in.asp

SQL Tutorial. (n.d.). *SQL INNER JOIN keyword*. w3schools.com. https://www.w3schools.com/sql/sql_join_inner.asp

SQL Tutorial. (n.d.). *SQL INSERT INTO statement*. w3schools.com. https://www.w3schools.com/sql/sql_insert.asp

SQL Tutorial. (n.d.). *SQL LEFT JOIN keyword*. w3schools.com. https://www.w3schools.com/sql/sql_join_left.asp

SQL Tutorial. (n.d.). *SQL LIKE operator*. w3schools.com. https://www.w3schools.com/sql/sql_like.asp

SQL Tutorial. (n.d.). *SQL MIN() and MAX() functions*. w3schools.com. https://www.w3schools.com/sql/sql_min_max.asp

SQL Tutorial. (n.d.). *SQL NULL values*. w3schools.com. https://www.w3schools.com/sql/sql_null_values.asp

SQL Tutorial. (n.d.). *SQL ORDER BY keyword*. w3schools.com. https://www.w3schools.com/sql/sql_orderby.asp

SQL Tutorial. (n.d.). *SQL SELECT statement*. w3schools.com. https://www.w3schools.com/sql/sql_select.asp

SQL Tutorial. (n.d.). *SQL UPDATE statement*. w3schools.com. https://www.w3schools.com/sql/sql_update.asp

SQL Tutorial. (n.d.). *SQL WHERE clause*. w3schools.com. https://www.w3schools.com/sql/sql_where.asp

SQL Tutorial. (n.d.). *SQL wildcards*. w3schools.com. https://www.w3schools.com/sql/sql_wildcards.asp

SQL Tutorial. (n.d.). *SQL working with dates*. w3schools.com. https://www.w3schools.com/sql/sql_dates.asp

SQLite. (n.d.). *Date and time functions*. https://sqlite.org/lang_datefunc.html

SQLite Tutorial. (n.d.). *SQLite ALTER TABLE*. https://www.sqlitetutorial.net/sqlite-alter-table/

SQLite Tutorial. (n.d.). *SQLite BETWEEN*. https://www.sqlitetutorial.net/sqlite-between/

SQLite Tutorial. (n.d.). *SQLite create table*. https://www.sqlitetutorial.net/sqlite-create-table/

SQLite Tutorial. (n.d.). *SQLite cross join*. SQLite Tutorial. https://www.sqlitetutorial.net/sqlite-cross-join/

SQLite Tutorial. (n.d.). *SQLite delete*. https://www.sqlitetutorial.net/sqlite-delete/

SQLite Tutorial. (n.d.). *SQLite drop table*. https://www.sqlitetutorial.net/sqlite-drop-table/

SQLite Tutorial. (n.d.). *SQLite group by*. https://www.sqlitetutorial.net/sqlite-group-by/

SQLite Tutorial. (n.d.). *SQLite having*. https://www.sqlitetutorial.net/sqlite-having/

SQLite Tutorial. (n.d.). *SQLite in*. https://www.sqlitetutorial.net/sqlite-in/

SQLite Tutorial. (n.d.). *SQLite inner join*. https://www.sqlitetutorial.net/sqlite-inner-join/

SQLite Tutorial. (n.d.). *SQLite insert*. https://www.sqlitetutorial.net/sqlite-insert/

SQLite Tutorial. (n.d.). *SQLite IS NULL*. https://www.sqlitetutorial.net/sqlite-is-null/

SQLite Tutorial. (n.d.). *SQLite left join*. https://www.sqlitetutorial.net/sqlite-left-join/

SQLite Tutorial. (n.d.). *SQLite like*. https://www.sqlitetutorial.net/sqlite-like/

SQLite Tutorial. (n.d.). *SQLite order by*. https://www.sqlitetutorial.net/sqlite-order-by/

SQLite Tutorial. (n.d.). *SQLite rename column*. https://www.sqlitetutorial.net/sqlite-rename-column/

SQLite Tutorial. (n.d.). *SQLite select*. https://www.sqlitetutorial.net/sqlite-select/

SQLite Tutorial. (n.d.). *SQLite union*. https://www.sqlitetutorial.net/sqlite-union/

SQLite Tutorial. (n.d.). *SQLite update*. https://www.sqlitetutorial.net/sqlite-update/

SQLite Tutorial. (n.d.). *SQLite where*. https://www.sqlitetutorial.net/sqlite-where/

w3resource. (n.d.). *SQL LOWER() function*. w3resource.com. https://www.w3resource.com/sql/character-functions/lower.php