Programming Fundamentals [1]

A Study Guide for Students of Sorsogon State University - Bulan Campus[2]

Jarrian Vince G. Gojar[3]

April 23, 2025

[1] A course in the Bachelor of Technical Vocational Teacher Education (BTVTED) Major in Computer System Servicing.

[2] This book is a study guide for students of Sorsogon State University - Bulan Campus taking up the course Programming Fundamentals.

[3] https://github.com/godkingjay

Sorsogon State University - Bulan Campus

Contents

Contents ii

List of Figures v

List of Tables vi

1 Introduction to Python 2
- 1.1 Introduction . . . 2
  - 1.1.1 History of Python . . . 2
  - 1.1.2 What is Programming Language? . . . 2
- 1.2 Environment Setup . . . 2
  - 1.2.1 How to Download and Install Python on Windows . . . 2
  - 1.2.2 How to Download PyCharm IDE on Windows . . . 4
- 1.3 First Python Program . . . 5
- 1.4 Basic Syntax . . . 5
  - 1.4.1 Variables . . . 5
  - 1.4.2 Python Identifiers . . . 6
  - 1.4.3 Reserved Words . . . 6
  - 1.4.4 Quotation in Python . . . 7
    - 1.4.4.1 Escape Characters . . . 7
    - 1.4.4.2 Multi-line Strings in Single or Double Quotes . . . 8
  - 1.4.5 Comments . . . 8
  - 1.4.6 Lines and Indentation . . . 9
  - 1.4.7 Multi-Line Statements . . . 9
  - 1.4.8 Data Types . . . 10
    - 1.4.8.1 Integers . . . 10
    - 1.4.8.2 Floats . . . 10
    - 1.4.8.3 Strings . . . 11
    - 1.4.8.4 Lists . . . 12
    - 1.4.8.5 Tuples . . . 12
    - 1.4.8.6 Dictionary . . . 13
  - 1.4.9 Conversion between Data Types . . . 14
  - 1.4.10 Basic Operators . . . 15
    - 1.4.10.1 Arithmetic Operators . . . 15
  - 1.4.11 PEMDAS Rule . . . 16
    - 1.4.11.1 Comparison Operators . . . 16
    - 1.4.11.2 Assignment Operators . . . 17
    - 1.4.11.3 Logical Operators . . . 18
- 1.5 Taking Input from the User . . . 19

2 Control Statements **21**
2.1 Introduction . . . 21
2.2 Conditional Statements . . . 21
2.2.1 If Statement . . . 21
2.2.2 If-Else Statement . . . 22
2.2.3 If-Elif-Else Statement . . . 23
2.3 Iterative Statements . . . 24
2.3.1 While Loop . . . 24
2.3.2 For Loop . . . 25
2.3.3 For Each Loop . . . 26
2.3.4 Nested Loops . . . 27
2.4 Jump Statements . . . 27
2.4.1 Break Statement . . . 27
2.4.2 Continue Statement . . . 28
2.4.3 Pass Statement . . . 29

3 Functions **30**
3.1 Introduction . . . 30
3.2 Defining Functions . . . 30
3.3 Calling Functions . . . 30
3.4 Parameters and Arguments . . . 31
3.5 Return Statement . . . 32
3.6 Recursion . . . 32
3.7 Lambda Functions . . . 33
3.8 Built-in Functions . . . 33
3.9 String Functions . . . 34
3.10 List Functions . . . 34

4 File Handling **36**
4.1 Introduction . . . 36
4.2 Opening and Closing Files . . . 36
4.3 Reading and Writing Files . . . 36
4.3.1 Reading Files . . . 36
4.3.1.1 `read()` Method . . . 37
4.3.1.2 `readline()` Method . . . 37
4.3.1.3 `readlines()` Method . . . 37
4.3.2 Writing Files . . . 39
4.3.2.1 `write()` Method . . . 39
4.3.2.2 `writelines()` Method . . . 39
4.4 CSV File . . . 40
4.4.1 Reading CSV Files . . . 41
4.4.1.1 Reading CSV Files with `csv.reader()` . . . 41
4.4.1.2 Reading CSV Files with `csv.DictReader()` . . . 41
4.4.2 Writing CSV Files . . . 43
4.4.2.1 Writing CSV Files with `csv.writerow()` . . . 43
4.4.2.2 Writing CSV Files with `csv.writerows()` . . . 44

5 Exception Handling **46**
5.1 Introduction . . . 46
5.2 Try-Except Block . . . 46
5.3 Finally Block . . . 47
5.4 Types of Exceptions . . . 47

5.5 Raising Exceptions . . . 48
5.6 User-Defined Exceptions . . . 48

6 Object-Oriented Programming 50
6.1 Introduction . . . 50
6.2 Classes and Objects . . . 50
6.3 Inheritance . . . 50
6.4 Polymorphism . . . 50
6.5 Encapsulation . . . 50
6.6 Abstraction . . . 50

7 Modules and Packages 51
7.1 Introduction . . . 51
7.2 Creating Modules . . . 51
7.3 Importing Modules . . . 51
7.4 Creating Packages . . . 51
7.5 Importing Packages . . . 51

8 Regular Expressions 52
8.1 Introduction . . . 52
8.2 Match Function . . . 52
8.3 Search Function . . . 52
8.4 Findall Function . . . 52
8.5 Split Function . . . 52
8.6 Sub Function . . . 52

9 Database Connectivity 53
9.1 Introduction . . . 53
9.2 Connecting to a Database . . . 53
9.3 Creating a Table . . . 53
9.4 Inserting Data . . . 53
9.5 Selecting Data . . . 53
9.6 Updating Data . . . 53
9.7 Deleting Data . . . 53

10 Graphical User Interface 54

11 References 55

List of Figures

1 Python Installation Wizard 3
2 Python Installation Verification 3
3 PyCharm Community Edition Download 4
4 PyCharm Community Edition Installation Wizard 4

List of Tables

4.1 Sample CSV File Content 41
4.2 Table View of CSV File Content 44

List of Codes

1.1 Hello, World! Program . . . 5
1.2 Variable Assignment . . . 5
1.3 Valid and Invalid Identifiers . . . 6
1.4 Quotation in Python . . . 7
1.5 Escape Characters in Python . . . 8
1.6 Multi-line Strings in Single or Double Quotes . . . 8
1.7 Comments in Python . . . 8
1.8 Lines and Indentation in Python . . . 9
1.9 Multi-Line Statements in Python . . . 9
1.10 Integers in Python . . . 10
1.11 Floats in Python . . . 11
1.12 Strings in Python . . . 11
1.13 Lists in Python . . . 12
1.14 Tuples in Python . . . 12
1.15 Dictionaries in Python . . . 13
1.16 Conversion between Data Types in Python . . . 14
1.17 Arithmetic Operators in Python . . . 15
1.18 PEMDAS Rule in Python . . . 16
1.19 Comparison Operators in Python . . . 16
1.20 Assignment Operators in Python . . . 17
1.21 Logical Operators in Python . . . 18
1.22 Taking Input from the User in Python . . . 19
2.1 Syntax of the If Statement . . . 21
2.2 If Statement in Python . . . 21
2.3 Syntax of the If-Else Statement . . . 22
2.4 If-Else Statement in Python . . . 22
2.5 Syntax of the If-Elif-Else Statement . . . 23
2.6 If-Elif-Else Statement in Python . . . 23
2.7 Syntax of the While Loop . . . 24
2.8 While Loop in Python . . . 24
2.9 Syntax of the For Loop . . . 25
2.10 For Loop in Python . . . 25
2.11 For Loop with List in Python . . . 25
2.12 For Each Loop in Python . . . 26
2.13 Nested Loops in Python . . . 27
2.14 Break Statement in Python . . . 27
2.15 Continue Statement in Python . . . 28
2.16 Pass Statement in Python . . . 29
3.1 Defining a Function in Python . . . 30
3.2 Calling a Function in Python . . . 30
3.3 Parameters and Arguments in Python . . . 31

3.4 Sum of Two Numbers in Python . . . 31
3.5 Recursion in Python . . . 33
3.6 Lambda Functions in Python . . . 33
3.7 String Functions in Python . . . 34
3.8 List Functions in Python . . . 35
4.1 Opening and Closing Files in Python . . . 36
4.2 Reading Files with read() Method in Python . . . 37
4.3 Reading Files with readline() Method in Python . . . 37
4.4 Reading Files with readlines() Method in Python . . . 37
4.5 Printing Each Line in the List . . . 38
4.6 Writing Files with write() Method in Python . . . 39
4.7 Writing Files in Append Mode . . . 39
4.8 Writing Files with writelines() Method in Python . . . 39
4.9 Sample CSV File Content . . . 40
4.10 Reading CSV Files with csv.reader() in Python . . . 41
4.11 Reading CSV Files with csv.DictReader() in Python . . . 42
4.12 Writing CSV Files with csv.writer() in Python . . . 43
4.13 Writing CSV Files with csv.writerow() in Python . . . 43
4.14 Writing CSV Files with csv.writerows() in Python . . . 44
5.1 Try-Except Block in Python . . . 46
5.2 Finally Block in Python . . . 47
5.3 Raising Exceptions in Python . . . 48
5.4 User-Defined Exceptions in Python . . . 48

Preface

> *"Programming is the art of telling another human being what one wants the computer to do."*
>
> – Donald Ervin Knuth

Jarrian Vince G. Gojar
https://github.com/godkingjay

1

Introduction to Python

1.1 Introduction

Python is a high-level, interpreted, interactive, and object-oriented programming language. It is designed to be highly readable, using English keywords frequently, unlike other languages that use punctuation, and it has fewer syntactical constructions. It is a versatile, general-purpose, and powerful programming language. It is an excellent first language because it is concise and easy to read. Whether you want to do web development, machine learning, or data science, Python is the language for you.

1.1.1 History of Python

Python was developed by **Guido van Rossum** in the late 80's and early 90's at the National Research Institute for Mathematics and Computer Science in the Netherlands. Python is derived from many other languages, including ABC, Modula-3, C, C++, Algol-68, SmallTalk, and Unix shell and other scripting languages.

1.1.2 What is Programming Language?

A programming language is a formal language comprising a set of instructions that produce various kinds of output. Programming languages are used in computer programming to implement algorithms. A computer only understands machine code, which is a series of binary numbers. Programming languages allow humans to write code that can be understood by computers. Python in particular is an interpreted language, which means that the code is executed line by line.

1.2 Environment Setup

In software development, an **integrated development environment (IDE)** is a software application that provides comprehensive facilities to computer programmers for software development. An IDE normally consists of a source code editor, build automation tools, and a debugger. Other than the IDE, it is also important to have the programming language installed on your computer.

1.2.1 How to Download and Install Python on Windows

To download and install Python on Windows, follow these steps:

1. Open a web browser and go to the official Python website at https://www.python.org/downloads/.
2. Click on the latest version of Python to download the installer.
3. Run the installer and follow the installation wizard.
4. Make sure to check the box that says "Add Python to PATH".
5. Click on the "Install Now" button to install Python on your computer.

Figure 1: Python Installation Wizard

6. Once the installation is complete, you can verify the installation by opening a command prompt and typing `python -version`.

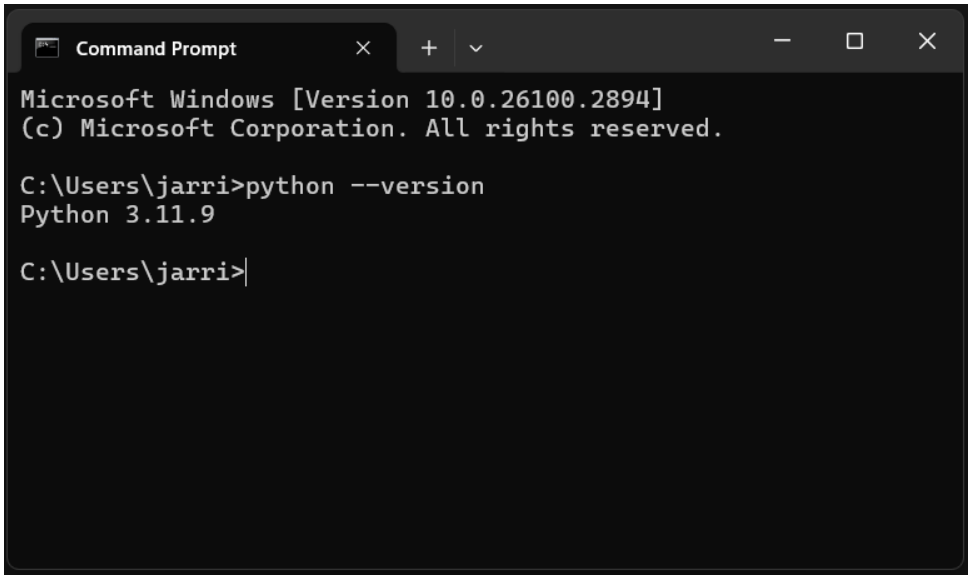

Figure 2: Python Installation Verification

1.2.2 How to Download PyCharm IDE on Windows

To download and install PyCharm IDE on Windows, follow these steps:

1. Open a web browser and go to the official PyCharm website at https://www.jetbrains.com/pycharm/download/.
2. Click on the "Download" button to download the installer.

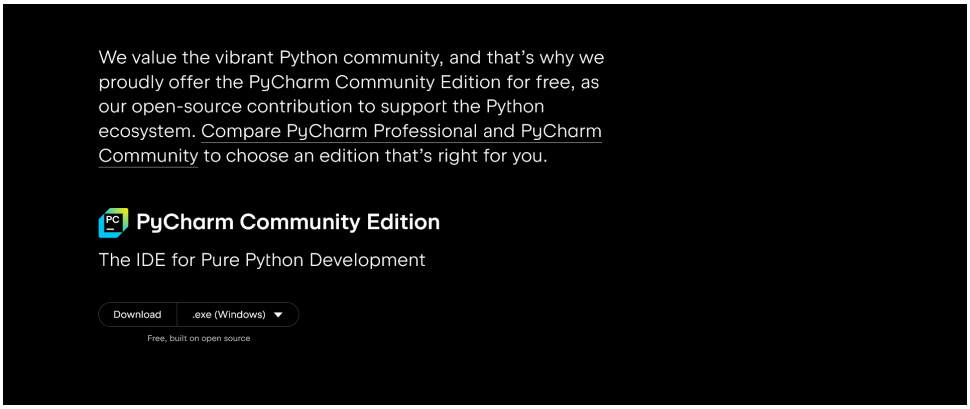

Figure 3: PyCharm Community Edition Download

3. Run the installer and follow the installation wizard.
4. Make sure to check the box that says "Create associations".
5. Click on the "Install" button to install PyCharm on your computer.

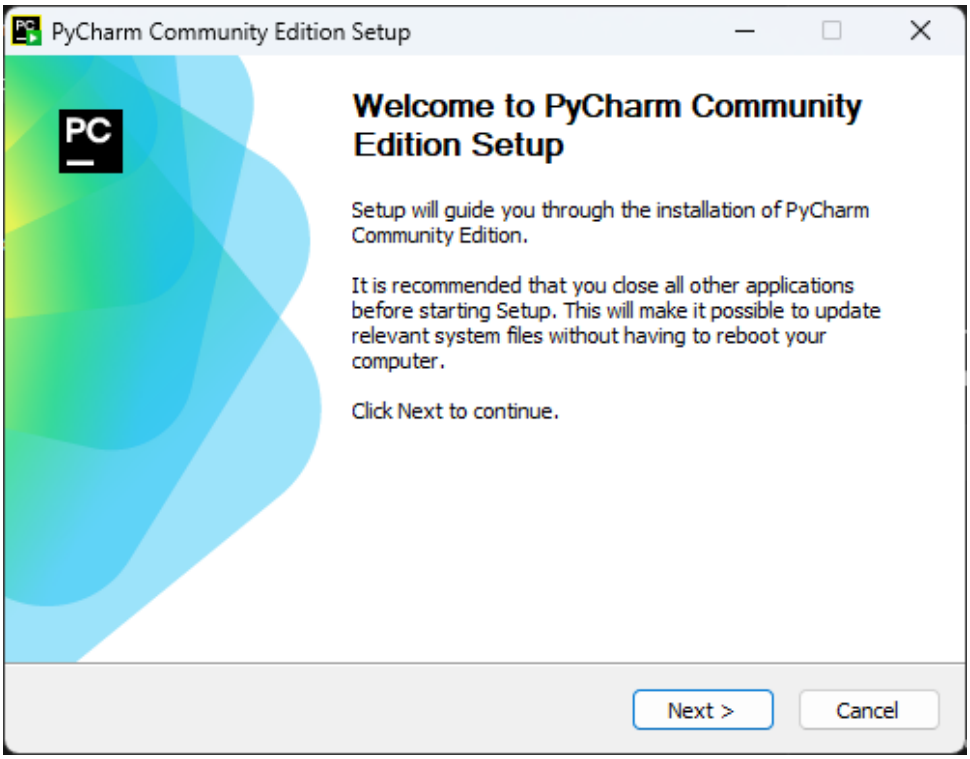

Figure 4: PyCharm Community Edition Installation Wizard

6. Once the installation is complete, you can launch PyCharm by clicking on the desktop shortcut or searching for it in the Start menu.

1.3 First Python Program

In Python, the `print()` function is used to display output on the screen. To write your first Python program, follow these steps:

1. Open PyCharm IDE on your computer.
2. Click on the "Create New Project" button to create a new project.
3. Enter a name for your project and click on the "Create" button.
4. Right-click on the project folder in the Project view and select "New" → "Python File".
5. Enter a name for your Python file and click on the "OK" button.
6. Type the following code in the Python file:

```
1    print("Hello, World!")
```

Code 1.1: Hello, World! Program

7. Click on the "Run" button to run the program.
8. You should see the output "Hello, World!" displayed in the Run window.

1.4 Basic Syntax

In Python, the syntax refers to the rules that define the combinations of symbols that are considered to be correctly structured programs in the language. The Python syntax is simple and easy to learn. It is based on indentation and does not require the use of curly braces or semicolons. It can also be easily understood by beginners as it is very similarly structured to the English language.

1.4.1 Variables

A variable is a name that refers to a value. In Python, variables are created when you assign a value to them. You can assign a value to a variable using the assignment operator `=`.

```
1  x = 5
2  y = "Hello, World!"
3
4  print(x)
5  print(y)
```

Code 1.2: Variable Assignment

Code 1.2 shows how to assign values to variables in Python. In this example, the variable `x` is assigned the value 5, and the variable `y` is assigned the value "Hello, World!".

Exercises

```
1. Create a variable called ‘name‘ and assign the value to your name.
2. Create a variable called ‘age‘ and assign the value to your age.
3. Create a variable called ‘address‘ and assign the value to your address.
4. Print the values of the variables ‘name‘, ‘age‘, and ‘address‘.
```

1.4.2 Python Identifiers

An identifier is a name given to entities like class, functions, variables, etc. It helps to differentiate one entity from another. In Python, an identifier is a name used to identify a variable, function, class, module, or other object. An identifier must start with a letter or an underscore (_), followed by letters, digits, or underscores.

```
1 # Valid Identifiers
2 my_variable = 5
3 myVariable = 10
4 _my_variable = 15
5
6 # Invalid Identifiers
7 1variable = 20
8 my-variable = 25
9 my variable = 30
```

Code 1.3: Valid and Invalid Identifiers

Code 1.3 shows examples of valid and invalid identifiers in Python. In this example, the variables `my_variable`, `myVariable`, and `_my_variable` are valid identifiers, while the variables `1variable`, `my-variable`, and `my variable` are invalid identifiers.

Exercises

```
1. Create a variable called ‘first_name‘ and assign the value to your first
   name.
2. Create a variable called ‘last_name‘ and assign the value to your last
   name.
3. Create a variable called ‘full_name‘ and assign the value of concatenating
   ‘first_name‘ and ‘last_name‘.
4. Print the value of the variable ‘full_name‘.
```

1.4.3 Reserved Words

Python has a set of reserved words that cannot be used as identifiers. These reserved words are used by the Python interpreter to recognize the structure of the program. Some of the reserved words in Python include:

- `False`
- `None`
- `True`
- `and`
- `as`
- `assert`
- `break`
- `class`
- `continue`
- `def`
- `del`
- `elif`
- `else`
- `except`
- `exec`
- `finally`
- `for`
- `from`
- `global`
- `if`
- `import`
- `in`
- `is`
- `lambda`
- `nonlocal`
- `not`
- `or`
- `pass`
- `print`
- `raise`
- `return`
- `try`
- `while`
- `with`
- `yield`

These reserved words cannot be used as identifiers in Python. They cannot be used as variable names, function names, or any other identifier names.

1.4.4 Quotation in Python

In Python, you can use either single quotes (’), double quotes ("), or triple quotes ("’ or """). Single and double quotes are used to represent strings, while triple quotes are used to represent multi-line strings.

```
single_quote = ’Hello, World!’
double_quote = "Hello, World!"
triple_quote = ’’’Hello,
World!’’’

print(single_quote)
print(double_quote)
print(triple_quote)
```

Code 1.4: Quotation in Python

Code 1.4 shows examples of using single quotes, double quotes, and triple quotes in Python. In this example, the variables `single_quote`, `double_quote`, and `triple_quote` are assigned string values using single quotes, double quotes, and triple quotes, respectively. A single quote or double quote can be used to represent a string value, while triple quotes are used to represent multi-line strings.

1.4.4.1 Escape Characters

An escape character is a backslash (\) followed by a character that has a special meaning in Python. It is used to represent characters that are difficult or impossible to type directly. Some of the common escape characters in Python include:

- `\n` - New Line
- `\t` - Tab
- `\r` - Carriage Return
- `\\` - Backslash
- `\’` - Single Quote
- `\"` - Double Quote
- `\b` - Backspace
- `\f` - Form Feed

These escape characters are used to represent special characters in Python. For example, the escape character `\n` is used to represent a newline character, while the escape character `\t` is used to represent a tab character.

```
new_line = "Hello,\nWorld!"
tab_space = "Hello,\tWorld!"
carriage_return = "Hello,\rWorld!"
back_slash = "Hello,\\World!"
single_quote = "Hello,\'World!"
double_quote = "Hello,\"World!"
back_space = "Hello,\bWorld!"
form_feed = "Hello,\fWorld!"

print(new_line)
print(tab_space)
print(carriage_return)
print(back_slash)
print(single_quote)
print(double_quote)
print(back_space)
print(form_feed)
```

Code 1.5: Escape Characters in Python

Code 1.5 shows examples of using escape characters in Python. In this example, the variables `new_line`, `tab_space`, `back_slash`, `single_quote`, `double_quote`, `back_space`, and `form_feed` are assigned string values using escape characters to represent special characters.

1.4.4.2 Multi-line Strings in Single or Double Quotes

In Python, you can use triple quotes (`"'` or `"""`) to represent multi-line strings. This allows you to write strings that span multiple lines without using escape characters. However, if you want to represent multi-line strings using single or double quotes, you can use the escape character `\n` to represent a newline character.

```
multi_line_single = 'Hello,\nWorld!'
multi_line_double = "Hello,\nWorld!"

print(multi_line_single)
print(multi_line_double)
```

Code 1.6: Multi-line Strings in Single or Double Quotes

Code 1.6 shows examples of using multi-line strings in single or double quotes in Python. In this example, the variables `multi_line_single` and `multi_line_double` are assigned string values using the escape character `\n` to represent a newline character.

1.4.5 Comments

Comments are used to explain the code and make it more readable. In Python, comments start with the hash character (`#`) and continue to the end of the line. Comments are ignored by the Python interpreter and are not executed as part of the program.

```
# This is a single-line comment

'''
This is a multi-line comment.
It can span multiple lines.
'''

"""
This is also a multi-line comment.
It can span multiple lines.
"""
```

Code 1.7: Comments in Python

Code 1.7 shows examples of single-line and multi-line comments in Python. In this example, the single-line comment starts with the hash character (`#`), while the multi-line comment is enclosed in triple quotes (`'''` or `"""`).

1.4.6 Lines and Indentation

Python uses indentation to define the structure of the code. Indentation is used to group statements together. The number of spaces in the indentation is not fixed, but all statements within the block must be indented the same amount.

```
if 5 > 2:
    print("Five is greater than two!")
```

Code 1.8: Lines and Indentation in Python

Code 1.8 shows an example of using indentation in Python. In this example, the `print()` statement is indented to indicate that it is part of the `if` block. The number of spaces in the indentation is not fixed, but all statements within the block must be indented the same amount.

1.4.7 Multi-Line Statements

In Python, a statement can span multiple lines if it is enclosed in parentheses `()`, square brackets `[]`, or curly braces `{}`. This is useful when you have a long statement that you want to split into multiple lines for readability.

```
numbers = [1, 2, 3, 4, 5,
           6, 7, 8, 9, 10]

total = (1 + 2 + 3 +
           4 + 5 + 6 +
           7 + 8 + 9 + 10)

colors = {'red': 255, 'green': 255,
           'blue': 255}
```

Code 1.9: Multi-Line Statements in Python

Code 1.9 shows examples of using multi-line statements in Python. In this example, the list `numbers`, the sum `total`, and the dictionary `colors` are defined using multi-line statements enclosed in square brackets, parentheses, and curly braces, respectively.

1.4.8 Data Types

In Python, every value has a data type. Data types are used to represent different types of data, such as numbers, strings, lists, tuples, dictionaries, etc. These data types are used to store, manipulate, and represent data in Python programs. Say for example, the data type `int` is used to represent integers, the data type `float` is used to represent floating-point numbers, and the data type `str` is used to represent strings. Using the correct data type is important because it determines how the data is stored and how it can be manipulated.

1.4.8.1 Integers

Integers are whole numbers, such as 1, 2, 3, 4, 5, etc. In Python, integers are represented using the `int` data type. Integers can be positive or negative, and they can be used in mathematical operations such as addition, subtraction, multiplication, and division.

```
x = 5
y = -10

print(x)
print(y)

print(type(x))
print(type(y))
```

Code 1.10: Integers in Python

Code 1.10 shows examples of using integers in Python. In this example, the variables `x` and `y` are assigned integer values 5 and -10, respectively. The `print()` function is used to display the values of the variables, and the `type()` function is used to display the data type of the variables. An integer is best used when you need to represent whole numbers without any decimal points like counting the number of students in a class or the number of books in a library.

Exercises

```
1. Create a variable called ‘age‘ and assign the value to your age.
2. Create a variable called ‘year‘ and assign the value to the current year.
3. Create a variable called ‘birth_year‘ and assign the value to 0.
4. Calculate the value of the variable ‘birth_year‘ by subtracting ‘age‘ from
   ‘year‘.
5. Print the value of the variable ‘birth_year‘.
```

1.4.8.2 Floats

Floats are decimal numbers, such as 1.0, 2.5, 3.14, 4.0, etc. In Python, floats are represented using the `float` data type. Floats can be used to represent real numbers, and they can be used in mathematical operations such as addition, subtraction, multiplication, and division.

```
x = 3.14
y = -2.5

print(x)
print(y)

print(type(x))
print(type(y))
```

Code 1.11: Floats in Python

Code 1.11 shows examples of using floats in Python. In this example, the variables `x` and `y` are assigned float values 3.14 and -2.5, respectively. A float is best used when you need to represent real numbers with decimal points like the price of an item or the temperature of a place.

Exercises

```
1. Create a variable called ‘price‘ and assign the value to the price of a
   product.
2. Create a variable called ‘quantity‘ and assign the value to the quantity
   of the product.
3. Create a variable called ‘total‘ and calculate the total cost of the
   product by multiplying ‘price‘ and ‘quantity‘.
4. Print the value of the variable ‘total‘.
```

1.4.8.3 Strings

Strings are sequences of characters, such as “Hello, World!”, “Python”, “Programming”, etc. In Python, strings are represented using the `str` data type. Strings can be enclosed in single quotes (’), double quotes ("), or triple quotes ("’ or """).

```
x = "Hello, World!"
y = 'Python'

print(x)
print(y)

print(type(x))
print(type(y))
```

Code 1.12: Strings in Python

Code 1.12 shows examples of using strings in Python. In this example, the variables `x` and `y` are assigned string values “Hello, World!” and “Python”, respectively. A string is best used when you need to represent text data like names, addresses, or messages.

Exercises

```
1. Create a variable called ‘first_name‘ and assign the value to your first
   name.
2. Create a variable called ‘last_name‘ and assign the value to your last
   name.
3. Create a variable called ‘full_name‘ and concatenate ‘first_name‘ and
   ‘last_name‘.
4. Print the value of the variable ‘full_name‘.
```

1.4.8.4 Lists

A list is a collection of items that are ordered and changeable. In Python, lists are represented using the `list` data type. Lists can contain items of different data types, such as integers, floats, strings, etc. Lists are enclosed in square brackets `[]` and the items are separated by commas.

```
numbers = [1, 2, 3, 4, 5]
fruits = ['apple', 'banana', 'cherry']

print(numbers)
print(fruits)

print(type(numbers))
print(type(fruits))
```

Code 1.13: Lists in Python

Code 1.13 shows examples of using lists in Python. In this example, the variables `numbers` and `fruits` are assigned list values `[1, 2, 3, 4, 5]` and `['apple', 'banana', 'cherry']`, respectively. A list is best used when you need to store multiple items in a single variable that can be changed like a list of numbers or a list of names.

Exercises

```
1. Create a list called ‘students‘ and assign the value to the names of your
   classmates.
2. Create a list called ‘grades‘ and assign the value to the grades of your
   classmates.
3. Print the values of the lists ‘students‘ and ‘grades‘.
```

1.4.8.5 Tuples

A tuple is a collection of items that are ordered and unchangeable. In Python, tuples are represented using the `tuple` data type. Tuples can contain items of different data types, such as integers, floats, strings, etc. Tuples are enclosed in parentheses `()` and the items are separated by commas.

```
coordinates = (1, 2, 3)
colors = ('red', 'green', 'blue')
```

```
3
4 print(coordinates)
5 print(colors)
6
7 print(type(coordinates))
8 print(type(colors))
```

Code 1.14: Tuples in Python

Code 1.14 shows examples of using tuples in Python. In this example, the variables `coordinates` and `colors` are assigned tuple values `(1, 2, 3)` and `('red', 'green', 'blue')`, respectively. A tuple is best used when you need to store multiple items in a single variable that should not be changed like the coordinates of a point or the RGB values of a color.

Exercises

```
1. Create a tuple called ‘point‘ and assign the value to the coordinates of a
   point.
2. Create a tuple called ‘rgb‘ and assign the value to the RGB values of a
   color.
3. Print the values of the tuples ‘point‘ and ‘rgb‘.
```

1.4.8.6 Dictionary

A dictionary is a collection of items that are unordered, changeable, and indexed. In Python, dictionaries are represented using the `dict` data type. Dictionaries consist of key-value pairs, where each key is associated with a value. Dictionaries are enclosed in curly braces `{}` and the key-value pairs are separated by commas.

```
1 person = {'name': 'John', 'age': 30, 'city': 'New York'}
2 colors = {'red': 255, 'green': 255, 'blue': 255}
3
4 print(person)
5 print(colors)
6
7 print(type(person))
8 print(type(colors))
```

Code 1.15: Dictionaries in Python

Code 1.15 shows examples of using dictionaries in Python. In this example, the variables `person` and `colors` are assigned dictionary values `{'name': 'John', 'age': 30, 'city': 'New York'}` and `{'red': 255, 'green': 255, 'blue': 255}`, respectively. A dictionary is best used when you need to store key-value pairs like the details of a person or the RGB values of a color.

1.4.9 Conversion between Data Types

In Python, you can convert one data type to another using built-in functions. Some of the common functions used for data type conversion include:

- `int()` - Converts a value to an integer.
- `float()` - Converts a value to a float.
- `str()` - Converts a value to a string.
- `list()` - Converts a value to a list.
- `tuple()` - Converts a value to a tuple.
- `dict()` - Converts a value to a dictionary.
- `set()` - Converts a value to a set.
- `bool()` - Converts a value to a boolean.
- `chr()` - Converts an integer to a character.

```
x = 5
y = 3.14
z = '10'

list_numbers = [1, 2, 3, 4, 5, 4, 3, 2, 1]
tuple_numbers = (1, 2, 3, 4, 5)
dict_colors = {'red': 255, 'green': 255, 'blue': 255}

print("Integer to Float:", float(x))
print("Float to Integer:", int(y))
print("String to Integer:", int(z))
print("List to Tuple:", tuple(list_numbers))
print("Tuple to List:", list(tuple_numbers))
print("Dictionary to List:", list(dict_colors))
print("List to Set:", set(list_numbers))
print("Integer to Boolean:", bool(x))
print("Integer to Character:", chr(65))
```

Code 1.16: Conversion between Data Types in Python

Code 1.16 shows examples of converting between data types in Python. In this example, the variables `x`, `y`, and `z` are assigned integer, float, and string values, respectively. The variables `list_numbers`, `tuple_numbers`, and `dict_colors` are assigned list, tuple, and dictionary values, respectively. The built-in functions are used to convert the values of these variables to different data types.

Exercises

```
1. Create a variable called ‘number‘ and assign the value to 10.
2. Create a variable called ‘price‘ and assign the value to 9.99.
3. Create a variable called ‘quantity‘ and assign the value to 5.
4. Convert the value of the variable ‘number‘ to a float and assign it to a
   new variable called ‘amount‘.
5. Convert the value of the variable ‘price‘ to an integer and assign it to a
   new variable called ‘total‘.
6. Convert the value of the variable ‘quantity‘ to a string and assign it to
```

```
   a new variable called ‘quantity_str‘.
7. Print the values of the variables ‘amount‘, ‘total‘, and ‘quantity_str‘.
```

1.4.10 Basic Operators

Operators are used to perform operations on variables and values. Python supports a wide range of operators, including arithmetic operators, comparison operators, assignment operators, logical operators, etc.

1.4.10.1 Arithmetic Operators

Arithmetic operators are used to perform mathematical operations such as addition, subtraction, multiplication, division, etc. Some of the common arithmetic operators in Python include:

- `+` - Addition
- `-` - Subtraction
- `*` - Multiplication
- `/` - Division
- `%` - Modulus
- `**` - Exponentiation
- `//` - Floor Division

```python
x = 10
y = 3

print("Addition:\t", x + y)
print("Subtraction:\t", x - y)
print("Multiplication:\t", x * y)
print("Division:\t", x / y)
print("Modulus:\t", x % y)
print("Exponentiation:\t", x ** y)
print("Floor Division:\t", x // y)
```

Code 1.17: Arithmetic Operators in Python

Code 1.17 shows examples of using arithmetic operators in Python. In this example, the variables `x` and `y` are assigned integer values 10 and 3, respectively. The arithmetic operators are used to perform addition, subtraction, multiplication, division, modulus, exponentiation, and floor division operations on these variables.

Exercises

```
1. Create a variable called ‘length‘ and assign the value to 10.
2. Create a variable called ‘width‘ and assign the value to 5.
3. Calculate the area of a rectangle by multiplying ‘length‘ and ‘width‘ and
   assign it to a new variable called ‘area‘.
4. Calculate the perimeter of a rectangle by adding the sum of ‘length‘ and
   ‘width‘ and multiply it by 2 and assign it to a new variable called
   ‘perimeter‘.
5. Print the values of the variables ‘area‘ and ‘perimeter‘.
```

1.4.11 PEMDAS Rule

In mathematics, the **order of operations** is a collection of rules that define the order in which different operations should be performed when evaluating an expression. The order of operations is commonly remembered using the acronym **PEMDAS**, which stands for:

- **P**arentheses
- **E**xponents
- **M**ultiplication and **D**ivision
- **A**ddition and **S**ubtraction

The PEMDAS rule is used to determine the order in which arithmetic operations should be performed in an expression. For example, in the expression $5 + 3 \times 2$, the multiplication should be performed first according to the PEMDAS rule, resulting in the value 11.

```
res1 = 5 + 3 * 2
res2 = (5 + 3) * 2

print("Result 1:", res1)
print("Result 2:", res2)
```

Code 1.18: PEMDAS Rule in Python

Code 1.18 shows examples of using the PEMDAS rule in Python. In this example, the expressions $5 + 3 \times 2$ and $(5 + 3) \times 2$ are evaluated using the arithmetic operators. The result of the first expression is 11, while the result of the second expression is 16. This demonstrates the importance of following the PEMDAS rule when evaluating arithmetic expressions.

Exercises

```
1. Create a variable called ‘result1‘ and assign the value to 10 + 5 * 2.
2. Create a variable called ‘result2‘ and assign the value to (10 + 5) * 2.
3. Create a variable called ‘result3‘ and assign the value to 10 + 5 ** 2.
4. Create a variable called ‘result4‘ and assign the value to (10 + 5) ** 2.
5. Print the values of the variables ‘result1‘, ‘result2‘, ‘result3‘, and
   ‘result4‘.
```

1.4.11.1 Comparison Operators

Comparison operators are used to compare two values and determine the relationship between them. Comparison operators return a boolean value `True` or `False` based on the comparison result. Some of the common comparison operators in Python include:

- `==` - Equal
- `!=` - Not Equal
- `>` - Greater Than
- `<` - Less Than
- `>=` - Greater Than or Equal To
- `<=` - Less Than or Equal To

```
x = 10
y = 5
```

```
3
4 print("Equal:\t", x == y)
5 print("Not Equal:\t", x != y)
6 print("Greater Than:\t", x > y)
7 print("Less Than:\t", x < y)
8 print("Greater Than or Equal To:\t", x >= y)
9 print("Less Than or Equal To:\t", x <= y)
```

Code 1.19: Comparison Operators in Python

Code 1.19 shows examples of using comparison operators in Python. In this example, the variables `x` and `y` are assigned integer values 10 and 5, respectively. The comparison operators are used to compare the values of these variables and return boolean values `True` or `False` based on the comparison result.

Exercises

```
1. Create a variable called ‘age‘ and assign the value to your age.
2. Create a variable called ‘adult‘ and assign the value to 18.
3. Create a variable called ‘is_adult‘ and compare ‘age‘ with ‘adult‘ using
    the greater than or equal to operator.
4. Print the value of the variable ‘is_adult‘.
```

1.4.11.2 Assignment Operators

Assignment operators are used to assign values to variables. They combine the assignment operator `=` with other operators to perform an operation and assign the result to a variable. Some of the common assignment operators in Python include:

- `+=` - Addition
- `-=` - Subtraction
- `*=` - Multiplication
- `/=` - Division
- `%=` - Modulus
- `**=` - Exponentiation
- `//=` - Floor Division

```
1 x = 10
2 y = 5
3
4 x += y
5 print("Addition:\t", x)
6
7 x -= y
8 print("Subtraction:\t", x)
9
10 x *= y
11 print("Multiplication:\t", x)
12
13 x /= y
14 print("Division:\t", x)
15
```

```
x %= y
print("Modulus:\t", x)

x **= y
print("Exponentiation:\t", x)

x //= y
print("Floor Division:\t", x)
```

Code 1.20: Assignment Operators in Python

Code 1.20 shows examples of using assignment operators in Python. In this example, the variables `x` and `y` are assigned integer values 10 and 5, respectively. The assignment operators are used to perform addition, subtraction, multiplication, division, modulus, exponentiation, and floor division operations on these variables and assign the result to the variable `x`.

Exercises

```
1. Create a variable called ‘total‘ and assign the value to 100.
2. Create a variable called ‘price‘ and assign the value to 10.
3. Subtract the value of the variable ‘price‘ from the variable ‘total‘ and
   assign it to the variable ‘balance‘.
4. Print the value of the variable ‘balance‘.
```

1.4.11.3 Logical Operators

Logical operators are used to combine multiple conditions and determine the relationship between them. Logical operators return a boolean value `True` or `False` based on the logical relationship between the conditions. Some of the common logical operators in Python include:

- `and` - Logical AND, returns `True` if both conditions are `True`.
- `or` - Logical OR, returns `True` if at least one condition is `True`.
- `not` - Logical NOT, returns the opposite of the condition.

```
x = 10
y = 5

print("Logical AND:\t", x > 5 and y < 10)
print("Logical OR:\t", x > 5 or y > 10)
print("Logical NOT:\t", not x > 5)
```

Code 1.21: Logical Operators in Python

Code 1.21 shows examples of using logical operators in Python. In this example, the variables `x` and `y` are assigned integer values 10 and 5, respectively. The logical operators are used to combine multiple conditions and determine the relationship between them, returning boolean values `True` or `False` based on the logical relationship.

Exercises

```
1. Create a variable called ‘age‘ and assign the value to your age.
2. Create a variable called ‘adult‘ and assign the value to 18.
3. Create a variable called ‘is_adult‘ and assign the value to ‘age‘ greater
   than or equal to ‘adult‘.
4. Create a variable called ‘is_teen‘ and assign the value to ‘age‘ less than
   18.
5. Print the values of the variables ‘is_adult‘ and ‘is_teen‘.
```

1.5 Taking Input from the User

In Python, you can take input from the user using the `input()` function. The `input()` function reads a line of text from the standard input device (usually the keyboard) and returns it as a string. You can use the `input()` function to prompt the user for input and store the result in a variable.

```
1 name = input("Enter your name: ")
2 age = int(input("Enter your age: "))
3
4 print("Name:", name)
5 print("Age:", age)
```

Code 1.22: Taking Input from the User in Python

Code 1.22 shows an example of taking input from the user in Python. In this example, the `input()` function is used to prompt the user for their name and age. The input values are stored in the variables `name` and `age`, respectively. The `print()` function is used to display the values of these variables.

Exercises

```
1. Greet the user.
   - Prompt the user to enter their first name and store it in a variable
     called ‘first_name‘.
   - Prompt the user to enter their last name and store it in a variable
     called ‘last_name‘.
   - Prompt the user to enter their age and store it in a variable called
     ‘age‘.
   - Print a message that says "Hello, <first_name> <last_name>! You are
     <age> years old.".

2. Simple Calculator.
   - Prompt the user to enter two numbers and store them in variables called
     ‘num1‘ and ‘num2‘.
   - Print the sum, difference, product, and quotient of the two numbers.

3. Favorite Color.
```

```
- Prompt the user to enter their favorite color and store it in a variable
    called ‘color‘.
- Print a message that says "<color> is a beautiful color.".
```

2

Control Statements

2.1 Introduction

Control statements are used to control the flow of a program based on certain conditions. They allow you to make decisions, repeat actions, and perform different operations based on the conditions specified in the program. Some of the common control statements in Python include:

- **Conditional Statements** - Used to make decisions based on certain conditions.
- **Iterative Statements** - Used to repeat actions multiple times based on certain conditions.
- **Jump Statements** - Used to transfer control from one part of the program to another.

2.2 Conditional Statements

Conditional statements are used to make decisions based on certain conditions. They allow you to execute different blocks of code based on the result of a condition. Conditional statements in Python include the `if`, `if-else`, and `if-elif-else` statements.

2.2.1 If Statement

The `if` statement is used to execute a block of code if a condition is `True`. If the condition is `False`, the block of code is not executed. The `if` statement is written using the `if` keyword followed by the condition and a colon `:`, and the block of code is indented.

```
if condition:
    # Block of code
```

Code 2.1: Syntax of the If Statement

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

if x > 0:
    print("The number is positive.")
```

Code 2.2: If Statement in Python

Code 2.2 shows an example of using the `if` statement in Python. In this example, the `input()` function is used to prompt the user for a number, which is stored in the variable `x`. The `if` statement is used to check if the number is positive, and if it is, a message is displayed indicating that the number is positive.

Exercises

```
1. Check if a number is even or odd.
    - Prompt the user to enter a number and store it in a variable called
      ‘number‘.
    - Use the ‘if‘ statement to check if the number is even or odd.
    - Print a message that says "The number is even." or "The number is odd.".

2. Check if a number is positive, negative, or zero.
    - Prompt the user to enter a number and store it in a variable called
      ‘number‘.
    - Use the ‘if‘ statement to check if the number is positive, negative, or
      zero.
    - Print a message that says "The number is positive.", "The number is
      negative.", or "The number is zero.".

3. Check if a student has passed or failed.
    - Prompt the user to enter the marks of a student and store it in a
      variable called ‘marks‘.
    - Use the ‘if‘ statement to check if the student has passed or failed.
    - Print a message that says "The student has passed." or "The student has
      failed.".
```

2.2.2 If-Else Statement

The `if-else` statement is used to execute one block of code if a condition is `True` and another block of code if the condition is `False`. The `if-else` statement is written using the `if` keyword followed by the condition and a colon `:`, the block of code to execute if the condition is `True`, the `else` keyword followed by a colon `:`, and the block of code to execute if the condition is `False`.

```
1 if condition:
2     # Block of code
3 else:
4     # Block of code
```

Code 2.3: Syntax of the If-Else Statement

```
1 x = int(input("Enter a number: "))
2 if x > 0:
3     print("The number is positive.")
4 else:
5     print("The number is not positive.")
```

Code 2.4: If-Else Statement in Python

Code 2.4 shows an example of using the `if-else` statement in Python. In this example, the `input()` function is used to prompt the user for a number, which is stored in the variable `x`. The `if-else` statement is used to check if the number is positive, and if it is, a message is displayed indicating that the number is positive; otherwise, a message is displayed indicating that the number is not positive.

2.2.3 If-Elif-Else Statement

The `if-elif-else` statement is used to execute one block of code if a condition is `True`, another block of code if another condition is `True`, and a default block of code if none of the conditions are `True`. The `if-elif-else` statement is written using the `if` keyword followed by the condition and a colon `:`, the block of code to execute if the condition is `True`, the `elif` keyword followed by the condition and a colon `:`, the block of code to execute if the condition is `True`, and the `else` keyword followed by a colon `:`, and the block of code to execute if none of the conditions are `True`.

```
if condition1:
    # Block of code
elif condition2:
    # Block of code
else:
    # Block of code
```

Code 2.5: Syntax of the If-Elif-Else Statement

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

if x > 0:
    print("The number is positive.")
elif x < 0:
    print("The number is negative.")
else:
    print("The number is zero.")
```

Code 2.6: If-Elif-Else Statement in Python

Code 2.6 shows an example of using the `if-elif-else` statement in Python. In this example, the `input()` function is used to prompt the user for a number, which is stored in the variable `x`. The `if-elif-else` statement is used to check if the number is positive, negative, or zero, and a message is displayed based on the result of the condition.

Exercises

```
1. Check if a number is divisible by 2, 3, or 5.
    - Prompt the user to enter a number and store it in a variable called
      ‘number‘.
    - Use the ‘if-elif-else‘ statement to check if the number is divisible by
      2, 3, or 5.
    - Print a message that says "The number is divisible by 2.", "The number
      is divisible by 3.", "The number is divisible by 5.", or "The number
```

```
       is not divisible by 2, 3, or 5.".

2. Check if a student has passed, failed, or needs improvement.
    - Prompt the user to enter the marks of a student and store it in a
       variable called `marks`.
    - Use the `if-elif-else` statement to check if the student has passed,
       failed, or needs improvement.
    - Print a message that says "The student has passed.", "The student has
       failed.", or "The student needs improvement.".

3. Check if a number is positive, negative, or zero using the `if-elif-else`
    statement.
    - Prompt the user to enter a number and store it in a variable called
       `number`.
    - Use the `if-elif-else` statement to check if the number is positive,
       negative, or zero.
    - Print a message that says "The number is positive.", "The number is
       negative.", or "The number is zero.".
```

2.3 Iterative Statements

Iterative statements are used to repeat actions multiple times based on certain conditions. They allow you to execute the same block of code repeatedly until a condition is met. Iterative statements in Python include the `for` loop and `while` loop.

2.3.1 While Loop

The `while` loop is used to execute a block of code as long as a condition is `True`. The `while` loop is written using the `while` keyword followed by the condition and a colon `:`, and the block of code to execute, which is indented.

```
while condition:
    # Block of code
```

Code 2.7: Syntax of the While Loop

```
i = 1
while i <= 5:
    print(i)
    i += 1
```

Code 2.8: While Loop in Python

Code 2.8 shows an example of using the `while` loop in Python. In this example, the variable `i` is assigned the value 1, and the `while` loop is used to print the value of `i` as long as `i` is less than or equal to 5. The value of `i` is incremented by 1 in each iteration of the loop.

Exercises

```
1. Numbers from 1 to 10
    - Use the `while` loop to print the numbers from 1 to 10.

2. Even Numbers from 1 to 10
    - Use the `while` loop to print the even numbers from 1 to 10.

3. Odd Numbers from 1 to 9
    - Use the `while` loop to print the odd numbers from 1 to 9.
```

2.3.2 For Loop

The `for` loop is used to iterate over a sequence of items such as a list, tuple, string, etc. The `for` loop is written using the `for` keyword followed by a variable to store the items, the `in` keyword, the sequence of items, and a colon `:`, and the block of code to execute, which is indented.

```
for item in sequence:
    # Block of code
```

Code 2.9: Syntax of the For Loop

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

Code 2.10: For Loop in Python

Code 2.10 shows an example of using the `for` loop in Python. In this example, the `range()` function is used to generate a sequence of numbers from 0 to 5, and the `for` loop is used to iterate over these numbers and print each number in the sequence.

```
cars = ['Toyota', 'Honda', 'Ford']

for i in range(len(cars)):
    print(i, cars[i])
```

Code 2.11: For Loop with List in Python

Code 2.11 shows another example of using the `for` loop in Python. In this example, the list `cars` contains three car names, and the `for` loop is used to iterate over the list and print the index and name of each car in the list.

Exercises

```
1. Even Numbers from 1 to 10
    - Use the `for` loop to print the even numbers from 1 to 10.
```

```
2. Iterate over a List
    - Create a list called ‘colors‘ and assign the value to a list of colors.
    - Use the ‘for‘ loop to iterate over the colors and print each color.

3. Multiplication Table
    - Prompt the user to enter a number and store it in a variable called
       ‘number‘.
    - Use the ‘for‘ loop to print the multiplication table of the number from
       1 to 10.

    Example:

        Enter a number: 5

        5 x 1 = 5
        5 x 2 = 10
        5 x 3 = 15
        5 x 4 = 20
        5 x 5 = 25
        5 x 6 = 30
        5 x 7 = 35
        5 x 8 = 40
        5 x 9 = 45
        5 x 10 = 50
```

2.3.3 For Each Loop

The `for each` loop is used to iterate over a sequence of items such as a list, tuple, string, etc., and access each item in the sequence. The `for each` loop is written using the `for` keyword followed by a variable to store the items, the `in` keyword, the sequence of items, and a colon `:`, and the block of code to execute, which is indented.

```
1 colors = ['red', 'green', 'blue']
2
3 for color in colors:
4     print(color)
```

Code 2.12: For Each Loop in Python

Code 2.12 shows an example of using the `for each` loop in Python. In this example, the list `colors` contains three color names, and the `for each` loop is used to iterate over the list and print each color name in the list.

Exercises

```
1. Characters of a String
    - Create a variable called ‘name‘ and assign the value to your name.
    - Use the ‘for each‘ loop to iterate over the characters of the string and
       print each character.
```

```
2. Items of a List
    - Create a list called ‘fruit‘ and assign the value to a list of fruits.
    - Use the ‘for each‘ loop to iterate over the items of the list and print
       each item.

3. Pass or Fail
    - Create a list called ‘marks‘ and assign the value to a list of marks.
    - Use the ‘for each‘ loop to iterate over the marks and check if the
       student has passed or failed.
    - Print a message that says "The student has passed." or "The student has
       failed.".

4. Even or Odd
    - Create a list called ‘numbers‘ and assign the value to a list of numbers.
    - Use the ‘for each‘ loop to iterate over the numbers and check if the
       number is even or odd.
    - Print a message that says "The number is even." or "The number is odd.".
```

2.3.4 Nested Loops

Nested loops are loops within loops. They allow you to iterate over multiple sequences of items by placing one loop inside another loop. Nested loops are useful when you need to perform operations on multiple sequences of items simultaneously.

```
1 for i in range(3):
2     for j in range(3):
3         print(i, j)
```

Code 2.13: Nested Loops in Python

Code 2.13 shows an example of using nested loops in Python. In this example, the outer loop iterates over the numbers 0 to 2, and the inner loop iterates over the numbers 0 to 2. The `print()` function is used to display the values of `i` and `j` in each iteration of the nested loops.

2.4 Jump Statements

Jump statements are used to transfer control from one part of the program to another. They allow you to change the flow of the program based on certain conditions. Jump statements in Python include the `break`, `continue`, and `pass` statements.

2.4.1 Break Statement

The `break` statement is used to exit a loop prematurely based on a certain condition. When the `break` statement is encountered, the loop is terminated, and the program continues with the next statement after the loop.

```
1 for i in range(10):
2     if i == 5:
```

```
3         break
4     print(i)
```

Code 2.14: Break Statement in Python

Code 2.14 shows an example of using the `break` statement in Python. In this example, the `for` loop iterates over the numbers 0 to 9, and the `if` statement checks if the value of `i` is equal to 5. When the value of `i` is equal to 5, the `break` statement is encountered, and the loop is terminated.

Exercises

```
1. Print the numbers from 1 to 10 using the ‘for‘ loop.
    - Use the ‘for‘ loop to print the numbers from 1 to 10.
    - Use the ‘break‘ statement to exit the loop when the number is 5.

2. Print the even numbers from 1 to 10 using the ‘for‘ loop.
    - Use the ‘for‘ loop to print the even numbers from 1 to 10.
    - Use the ‘break‘ statement to exit the loop when the number is 6.
```

2.4.2 Continue Statement

The `continue` statement is used to skip the current iteration of a loop based on a certain condition and continue with the next iteration. When the `continue` statement is encountered, the current iteration of the loop is skipped, and the program continues with the next iteration of the loop.

```
1 for i in range(10):
2     if i % 2 == 0:
3         continue
4     print(i)
```

Code 2.15: Continue Statement in Python

Code 2.15 shows an example of using the `continue` statement in Python. In this example, the `for` loop iterates over the numbers 0 to 9, and the `if` statement checks if the value of `i` is divisible by 2. When the value of `i` is divisible by 2, the `continue` statement is executed, and the current iteration of the loop is skipped.

Exercises

```
1. Print the even numbers from 1 to 10 using the ‘for‘ loop.
    - Use the ‘for‘ loop to print the even numbers from 1 to 10.
    - Use the ‘continue‘ statement to skip the odd numbers.

2. Print the numbers from 1 to 10 using the ‘for‘ loop.
    - Use the ‘for‘ loop to print the numbers from 1 to 10.
    - Use the ‘continue‘ statement to skip the number 5.
```

2.4.3 Pass Statement

The `pass` statement is used as a placeholder when no action is required. It is often used to define empty classes, functions, or loops that need to be implemented later. The `pass` statement does nothing and allows the program to continue without raising an error.

```
1 for i in range(10):
2     pass
```

Code 2.16: Pass Statement in Python

Code 2.16 shows an example of using the `pass` statement in Python. In this example, the `for` loop iterates over the numbers 0 to 9, and the `pass` statement is used as a placeholder inside the loop.

3

Functions

3.1 Introduction

Functions are blocks of code that perform a specific task or operation. They allow you to organize your code into reusable components that can be called multiple times. Functions help you avoid writing the same code multiple times and make your code more modular and easier to maintain.

3.2 Defining Functions

In Python, you can define a function using the `def` keyword followed by the function name, a pair of parentheses `()`, and a colon `:`. The block of code to be executed by the function is indented.

```
1 def greet():
2     print("Hello, World!")
```

Code 3.1: Defining a Function in Python

Code 3.1 shows an example of defining a function in Python. In this example, the `def` keyword is used to define a function called `greet`, which prints the message "Hello, World!" when called.

3.3 Calling Functions

To call a function in Python, you simply write the function name followed by a pair of parentheses `()`. When a function is called, the block of code inside the function is executed, and the result of the function is returned.

```
1 def greet():
2     print("Hello, World!")
3
4 greet()
```

Code 3.2: Calling a Function in Python

Code 3.2 shows an example of calling a function in Python. In this example, the function `greet` is defined to print the message "Hello, World!", and the function is called to execute the block of code inside the function.

Exercises

```
1. Greet the User
    - Define a function called ‘greet_user‘ that prints the message "Hello,
      User!".
    - Call the function to greet the user.

2. Print the Numbers from 1 to 10
    - Define a function called ‘print_numbers‘ that prints the numbers from 1
      to 10 using the ‘for‘ loop.
    - Call the function to print the numbers from 1 to 10.

3. Print the Even Numbers from 1 to 10
    - Define a function called ‘print_even_numbers‘ that prints the even
      numbers from 1 to 10 using the ‘for‘ loop.
    - Call the function to print the even numbers from 1 to 10.
```

3.4 Parameters and Arguments

Parameters are variables that are used to pass values to a function. When defining a function, you can specify one or more parameters inside the parentheses `()`. When calling a function, you can pass values to the parameters as **arguments**.

```
1 def greet(name):
2     print("Hello,", name)
3
4 greet("Alice")
```

Code 3.3: Parameters and Arguments in Python

Code 3.3 shows an example of using parameters and arguments in Python. In this example, the function `greet` is defined with a parameter `name`, and the function is called with the argument `"Alice"` to pass the value to the parameter.

```
1 def add(x, y):
2     return x + y
3
4 result = add(5, 3)
5 print("Sum:", result)
6
7 result = add(10, 7)
8 print("Sum:", result)
```

Code 3.4: Sum of Two Numbers in Python

Code 3.4 shows another example of using parameters and arguments in Python. In this example, the function `add` is defined with two parameters `x` and `y`, and the function is called with two arguments `5` and `3` to pass the values to the parameters.

Exercises

```
1. Calculator Functions
   - Define a function called ‘add‘ that takes two numbers as parameters and
     returns the sum of the numbers.
   - Define a function called ‘subtract‘ that takes two numbers as parameters
     and returns the difference of the numbers.
   - Define a function called ‘multiply‘ that takes two numbers as parameters
     and returns the product of the numbers.
   - Define a function called ‘divide‘ that takes two numbers as parameters
     and returns the quotient of the numbers.
   - Print a Selection Menu that allows the user to choose an operation (add,
     subtract, multiply, divide).

   example:

       Selection Menu
       1. Add
       2. Subtract
       3. Multiply
       4. Divide

       Enter your choice: 1
       Enter the first number: 5
       Enter the second number: 3
       Sum: 8

2. Greet the User with a Message
   - Define a function called ‘greet_user‘ that takes the user’s name as a
     parameter and prints the message "Hello, <name>!".
   - Prompt the user to enter their name and store it in a variable called
     ‘name‘.
   - Call the function to greet the user with the message "Hello, <name>!".
```

3.5 Return Statement

The `return` statement is a `jump` statement that is used to return a value from a function. When a `return` statement is encountered inside a function, the function is terminated, and the value is returned to the caller.

3.6 Recursion

Recursion is a programming technique where a function calls itself to solve a problem. Recursion is useful when a problem can be broken down into smaller subproblems that are similar to the original problem.

```python
def factorial(n):
    if n == 0:
        return 1
    else:
        return n * factorial(n - 1)

result = factorial(5)
print("Factorial:", result)
```

Code 3.5: Recursion in Python

Code 3.5 shows an example of using recursion in Python. In this example, the function `factorial` is defined to calculate the factorial of a number using recursion. The function calls itself to calculate the factorial of the number `n` by multiplying `n` with the factorial of `n - 1`.

Exercises

```
1. Fibonacci Series
    - Define a function called ‘fibonacci‘ that takes a number as a parameter
        and returns the Fibonacci series up to that number.
    - Print the Fibonacci series up to the number 10.

2. Sum of Digits
    - Define a function called ‘sum_of_digits‘ that takes a number as a
        parameter and returns the sum of the digits of the number.
    - Print the sum of the digits of the number 123.
```

3.7 Lambda Functions

Lambda functions are small, anonymous functions that can have any number of arguments but only one expression. They are defined using the `lambda` keyword followed by the arguments, a colon `:`, and the expression to be evaluated.

```python
add = lambda x, y: x + y
result = add(5, 3)
print("Sum:", result)
```

Code 3.6: Lambda Functions in Python

Code 3.6 shows an example of using lambda functions in Python. In this example, a lambda function called `add` is defined to take two arguments `x` and `y` and return the sum of the arguments. The lambda function is called with the arguments `5` and `3` to calculate the sum of the numbers.

3.8 Built-in Functions

Python provides a number of built-in functions that are available for use without the need to define them. Some of the common built-in functions in Python include:

- `print()` - Used to print messages to the console.
- `input()` - Used to take input from the user.
- `len()` - Used to get the length of a sequence.
- `range()` - Used to generate a sequence of numbers.
- `sum()` - Used to calculate the sum of a sequence of numbers.
- `max()` - Used to get the maximum value in a sequence.
- `min()` - Used to get the minimum value in a sequence.

3.9 String Functions

Python provides a number of built-in functions that are available for working with strings. Some of the common string functions in Python include:

- `upper()` - Used to convert a string to uppercase.
- `lower()` - Used to convert a string to lowercase.
- `capitalize()` - Used to capitalize the first letter of a string.
- `title()` - Used to capitalize the first letter of each word in a string.
- `strip()` - Used to remove leading and trailing whitespace from a string.
- `replace()` - Used to replace a substring in a string with another substring.
- `split()` - Used to split a string into a list of substrings.
- `join()` - Used to join a list of strings into a single string.

```
name = "alice smith"

print("Uppercase:", name.upper())
print("Lowercase:", name.lower())
print("Capitalized:", name.capitalize())
print("Title:", name.title())
print("Stripped:", name.strip())
print("Replaced:", name.replace("a", "o"))
print("Split:", name.split())
print("Joined:", "-".join(name.split()))
```

Code 3.7: String Functions in Python

Code 3.7 shows an example of using string functions in Python. In this example, the string `name` contains the value `"alice smith"`, and various string functions are used to manipulate the string and display the results.

3.10 List Functions

Python provides a number of built-in functions that are available for working with lists. Some of the common list functions in Python include:

- `append()` - Used to add an item to the end of a list.
- `insert()` - Used to insert an item at a specific position in a list.
- `remove()` - Used to remove an item from a list.
- `pop()` - Used to remove and return an item from a list.
- `clear()` - Used to remove all items from a list.
- `index()` - Used to get the index of an item in a list.

- `count()` - Used to count the number of occurrences of an item in a list.
- `sort()` - Used to sort the items in a list.
- `reverse()` - Used to reverse the order of items in a list.

```
numbers = [5, 3, 7, 2, 8]

numbers.append(4)
print("Appended:", numbers)

numbers.insert(2, 6)
print("Inserted:", numbers)

numbers.remove(7)
print("Removed:", numbers)

item = numbers.pop()
print("Popped:", item, numbers)

numbers.clear()
print("Cleared:", numbers)

index = numbers.index(3)
print("Index:", index)

count = numbers.count(5)
print("Count:", count)

numbers.sort()
print("Sorted:", numbers)

numbers.reverse()
print("Reversed:", numbers)
```

Code 3.8: List Functions in Python

Code 3.8 shows an example of using list functions in Python. In this example, the list `numbers` contains the values `[5, 3, 7, 2, 8]`, and various list functions are used to manipulate the list and display the results.

4

File Handling

4.1 Introduction

File handling is the process of working with files on a computer. In Python, you can read from and write to files using built-in functions and methods. File handling allows you to store and retrieve data from files on your computer.

4.2 Opening and Closing Files

To work with files in Python, you need to open the file using the `open()` function, which takes two arguments: the name of the file and the mode in which the file should be opened. The mode can be `"r"` for reading, `"w"` for writing, or `"a"` for appending.

```
file = open("data.txt", "w")

file.write("Hello, World!")
file.close()
```

Code 4.1: Opening and Closing Files in Python

Code 4.1 shows an example of opening and closing a file in Python. In this example, the `open()` function is used to open a file called `"data.txt"` in write mode, and the `write()` method is used to write the message `"Hello, World!"` to the file. The `close()` method is used to close the file after writing to it.

4.3 Reading and Writing Files

In Python, you can read from and write to files using the `read()`, `readline()`, and `readlines()` methods for reading, and the `write()` and `writelines()` methods for writing.

4.3.1 Reading Files

To read from a file, you can use the `read()` method to read the entire file, the `readline()` method to read a single line, or the `readlines()` method to read all lines into a list.

4.3.1.1 `read()` Method

The `read()` method is used to read the entire contents of a file.

```
file = open("data.txt", "r")

content = file.read()
print(content)
file.close()
```

Code 4.2: Reading Files with read() Method in Python

Code 4.2 shows an example of using the `read()` method to read the contents of a file in Python. In this example, the `open()` function is used to open a file called `"data.txt"` in read mode, and the `read()` method is used to read the entire contents of the file. The contents are then printed to the console. Finally, the `close()` method is used to close the file after reading.

4.3.1.2 `readline()` Method

The `readline()` method is used to read a single line from a file. Unlike the `read()` method, which reads the entire file, the `readline()` only reads one line at a time.

```
file = open("data.txt", "r")

line1 = file.readline()
print(line1)

line2 = file.readline()
print(line2)

line3 = file.readline()
print(line3)

file.close()
```

Code 4.3: Reading Files with readline() Method in Python

Code 4.3 shows an example of using the `readline()` method to read a single line from a file in Python. In this example, the `open()` function is used to open a file called `"data.txt"` in read mode, and the `readline()` method is used to read a single line from the file. The line is then printed to the console. Finally, the `close()` method is used to close the file after reading.

4.3.1.3 `readlines()` Method

The `readlines()` method is used to read all lines from a file into a list. Each line in the file is stored as a separate element in the list.

```
file = open("data.txt", "r")

lines = file.readlines()
```

```
print(lines)
file.close()
```

Code 4.4: Reading Files with readlines() Method in Python

Code 4.4 shows an example of using the `readlines()` method to read all lines from a file into a list in Python. In this example, the `open()` function is used to open a file called `"data.txt"` in read mode, and the `readlines()` method is used to read all lines from the file into a list. The list of lines is then printed to the console. Finally, the `close()` method is used to close the file after reading.

```
file = open("data.txt", "r")

lines = file.readlines()

for line in lines:
    print(line.strip())

file.close()
```

Code 4.5: Printing Each Line in the List

Code 4.5 shows an example of printing each line in the list. In this example, the `for` loop is used to iterate over the list of lines, and the `strip()` method is used to remove leading and trailing whitespace from each line before printing it to the console. Finally, the `close()` method is used to close the file after reading.

Exercises

```
1. Create a File
    - Create a file called ‘bm.txt‘ and write the following lines to the file:

        45
        +
        78

2. Read the File
    - Read the contents of the file ‘bm.txt‘.
    - Assign the first line to a variable called ‘num1‘.
    - Assign the second line to a variable called ‘operator‘.
    - Assign the third line to a variable called ‘num2‘.

3. Perform the Operation
    - Use the ‘if-elif-else‘ statement to check the value of ‘operator‘.
    - If ‘operator‘ is ‘+‘, add ‘num1‘ and ‘num2‘ and print the result.
    - If ‘operator‘ is ‘-‘, subtract ‘num2‘ from ‘num1‘ and print the result.
    - If ‘operator‘ is ‘*‘, multiply ‘num1‘ and ‘num2‘ and print the result.
    - If ‘operator‘ is ‘/‘, divide ‘num1‘ by ‘num2‘ and print the result.
    - If none of the above, print "Invalid operator".
```

4.3.2 Writing Files

To write to a file, you can use the `write()` method to write a string to a file, or the `writelines()` method to write a list of strings to a file.

4.3.2.1 `write()` Method

The `write()` method is used to write a string to a file. If the file is opened in write mode, the contents of the file will be overwritten. If the file is opened in append mode, the string will be added to the end of the file. If the file does not exist, it will be created.

```
file = open("data.txt", "w")

file.write("Hello, World!\n")
file.write("This is a test.\n")
file.write("Goodbye!\n")
file.close()
```

Code 4.6: Writing Files with write() Method in Python

Code 4.6 shows an example of using the `write()` method to write a string to a file in Python. In this example, the `write()` method is used to write three lines of text to the file.

The first line contains the message "Hello, World!", the second line contains the message "This is a test.", and the third line contains the message "Goodbye!". The `close()` method is used to close the file after writing.

```
file = open("data.txt", "a")

file.write("This is an appended line.\n")
file.close()
```

Code 4.7: Writing Files in Append Mode

Code 4.7 shows an example of writing to a file in append mode. Unlike the write mode, the append mode does not overwrite the existing contents of the file. Instead, it adds the new content to the end of the file.

4.3.2.2 `writelines()` Method

The `writelines()` method is used to write a list of strings to a file. Each string in the list will be written to the file as a separate line.

```
file = open("data.txt", "w")

lines = [
    "The quick brown fox jumps over the lazy dog.\n",
    "Python is a programming language.\n",
    "File handling is important.\n",
]

```

```
file.writelines(lines)
file.close()
```

Code 4.8: Writing Files with writelines() Method in Python

Code 4.8 shows an example of using the `writelines()`. In this example, the `writelines()` method is used to write a list of strings to the file. Each string in the list is written to the file as a separately.

Exercises

```
1. Create a File
   - Create a file called ‘my_information.txt‘.

2. Saving Information
   - Ask the user to enter their name, age, and favorite color.
   - Write the information to the file ‘my_information.txt‘ in the following
     format:

     Name: <name>
     Age: <age>
     Favorite Color: <color>
```

4.4 CSV File

CSV (Comma-Separated Values) files are used to store tabular data in a plain text format. Each line in a CSV file represents a row of data, and the values in each row are separated by commas. CSV files are commonly used for data exchange between applications and can be easily opened in spreadsheet software like Microsoft Excel or Google Sheets. CSV files are often used to store data in a structured format, making it easy to read and write data using Python's built-in CSV module.

```
id,name,age,address
1,John Doe,25,123 Main St
2,Jane Smith,30,456 Elm St
3,Bob Johnson,35,789 Oak St
4,Alice Brown,28,101 Pine St
5,Charlie Green,40,202 Maple St
```

Code 4.9: Sample CSV File Content

The above code shows an example of a CSV file content. In this example, the CSV file contains five rows of data, each representing a person's information. The first row contains the headers, which describe the data in each column. The subsequent rows contain the actual data for each person, including their ID, name, age, and address.

id	name	age	address
1	John Doe	25	123 Main St
2	Jane Smith	30	456 Elm St
3	Bob Johnson	35	789 Oak St
4	Alice Brown	28	101 Pine St
5	Charlie Green	40	202 Maple St

Table 4.1: Sample CSV File Content

Table 4.1 shows the same CSV file content in a tabular format. The table contains four columns: **id**, **name**, **age**, and **address**. Each row in the table represents a person's information, with the first row containing the headers that describe the data in each column.

4.4.1 Reading CSV Files

To read a CSV file in Python, you can use the `csv` module, which provides functions for reading and writing CSV files. The `csv.reader()` function is used to read a CSV file and return the data as a list of rows. Each row is represented as a list of values, and you can iterate over the rows to access the data. The `csv.DictReader()` function is used to read a CSV file and return the data as a list of dictionaries. Each row is represented as a dictionary, with the keys being the column headers and the values being the data in each column. The `csv.DictReader()` function is useful when you want to access the data by column name instead of by index.

4.4.1.1 Reading CSV Files with `csv.reader()`

The `csv.reader()` function is used to read a CSV file and return the data as a list of rows. Each row is represented as a list of values, and you can iterate over the rows to access the data.

```
1  import csv
2
3  file = open("data.csv", "r")
4
5  reader = csv.reader(file)
6
7  for row in reader:
8      print(row)
9
10 file.close()
```

Code 4.10: Reading CSV Files with csv.reader() in Python

Code 4.10 shows an example of using the `csv.reader()` function to read a CSV file in Python. First, the `csv` module is imported, and the `open()` function is used to open a file called `"data.csv"` in read mode. The `csv.reader()` function is then used to create a CSV reader object, which is used to read the contents of the file. The `for` loop is used to iterate over the rows in the CSV file, and each row is printed to the console. Finally, the `close()` method is used to close the file after reading.

4.4.1.2 Reading CSV Files with `csv.DictReader()`

The `csv.DictReader()` function is used to read a CSV file and return the data as a list of dictionaries. Each row is represented as a dictionary, the keys being the column headers and the

values being the data in each column. The `csv.DictReader()` function is useful when you want to access the data by column name instead of by index.

```
import csv

file = open("data.csv", "r")

reader = csv.DictReader(file)

for row in reader:
    print(row["id"], row["name"], row["age"], row["address"])

file.close()
```

Code 4.11: Reading CSV Files with csv.DictReader() in Python

Code 4.11 shows an example of using the `csv.DictReader()` function to read a CSV file in Python. In this example, the `csv.DictReader()` function is used to create a CSV reader object that will return the data as a list of dictionaries. For each row in the CSV file, the `id`, `name`, `age`, and `address` values are printed to the console using the keys of the dictionary.

Exercises

```
1. Create a CSV File
    - Create a CSV file called ‘grades.csv‘ with the following columns:
        - id
        - name
        - subject
        - grade
    - Add the following data to the CSV file:
        - 1, John Doe, Math, 74
        - 2, Jane Smith, Science, 80
        - 3, Bob Johnson, English, 90
        - 4, Alice Brown, History, 92
        - 5, Charlie Green, Geography, 86

2. Read the CSV File using DictReader
    - Read the contents of the CSV file ‘grades.csv‘ using the
      ‘csv.DictReader()‘ function.
    - Print the id, name, subject, and grade for each student in the CSV file
      using the following format:

        Student ID: 1
        Name: John Doe
        Subject: Math
        Grade: 74

        Student ID: 2
        Name: Jane Smith
        Subject: Science
```

```
Grade: 80

...
```

4.4.2 Writing CSV Files

To write to a CSV file in Python, you can use the `csv.writer()` function, which provides methods for writing rows of data to a CSV file. The `csv.writer()` function is used to create a CSV writer object, which is used to write the contents of the file. The `writerow()` method is used to write a single row of data to the CSV file, and the `writerows()` is used to write multiple rows of data to the CSV file. The `writerow()` method takes a list of values as an argument, and each value in the list is inserted into a separate column in the CSV file. The `writerows()` on the other hand, takes a list of lists as an argument, and each inner list is written as a separate row in the CSV file.

```
1  import csv
2
3  file = open("data.csv", "w", newline="")
4  writer = csv.writer(file)
5
6  writer.writerow(["id", "name", "age", "address"])
7  writer.writerow([1, "John Doe", 25, "123 Main St"])
8  writer.writerow([2, "Jane Smith", 30, "456 Elm St"])
9  writer.writerow([3, "Bob Johnson", 35, "789 Oak St"])
10 writer.writerow([4, "Alice Brown", 28, "101 Pine St"])
11 writer.writerow([5, "Charlie Green", 40, "202 Maple St"])
12
13 file.close()
```

Code 4.12: Writing CSV Files with csv.writer() in Python

Code 4.12 shows an example of using the `csv.writer()` function to write a CSV file in Python. In this example, the `csv` module is imported, and the `open()` function is used to open a file called
texttt"data.csv" in write mode. The **newline** argument is set to an empty string to prevent extra blank lines from being added to the file. The `csv.writer()` function is then used to create a CSV writer object, which is used to write the contents of the file. The `writerow()` method is used to write the header row and the data rows to the CSV file. Finally, the `close()` method is used to close the file after writing.

4.4.2.1 Writing CSV Files with `csv.writerow()`

To write a single row of data to a CSV file, you can use the `csv.writerow()` method. The `writerow()` method takes a list of values as an argument, and each value in the list is inserted into a separate column in the CSV file.

```
1  import csv
2
3  file = open("data.csv", "w", newline="")
```

```
writer = csv.writer(file)

writer.writerow(["id", "name", "age", "address"])
writer.writerow([1, "John Doe", 25, "123 Main St"])
writer.writerow([2, "Jane Smith", 30, "456 Elm St"])
writer.writerow([3, "Bob Johnson", 35, "789 Oak St"])
writer.writerow([4, "Alice Brown", 28, "101 Pine St"])
writer.writerow([5, "Charlie Green", 40, "202 Maple St"])

file.close()
```

Code 4.13: Writing CSV Files with csv.writerow() in Python

id	name	age	address
1	John Doe	25	123 Main St
2	Jane Smith	30	456 Elm St
3	Bob Johnson	35	789 Oak St
4	Alice Brown	28	101 Pine St
5	Charlie Green	40	202 Maple St

Table 4.2: Table View of CSV File Content

Code 4.13 shows an example of using the `csv.writerow()` method to write a single row of data to a CSV file in Python. In this example, the `writerow()` method is used to write the header row and the data rows to the CSV file. Each row is represented as a list of values, and the `writerow()` method is used to write each row to the CSV file. When the file is opened in write mode, the contents of the file will be overwritten. If the file is opened in append mode, the new rows will be added to the CSV file. Table 4.2 shows the same CSV file content in a tabular format after writing the data to the file.

4.4.2.2 Writing CSV Files with `csv.writerows()`

Similarly, to write multiple rows of data to a CSV file, you can use the `csv.writerows()` method. The `writerows()` method takes a list of lists as an argument, and each inner list is written as a separate row in the CSV file.

```
import csv

file = open("data.csv", "w", newline="")
writer = csv.writer(file)

writer.writerow(["id", "name", "age", "address"])
writer.writerows([
    [1, "John Doe", 25, "123 Main St"],
    [2, "Jane Smith", 30, "456 Elm St"],
    [3, "Bob Johnson", 35, "789 Oak St"],
    [4, "Alice Brown", 28, "101 Pine St"],
    [5, "Charlie Green", 40, "202 Maple St"]
])

file.close()
```

Code 4.14: Writing CSV Files with csv.writerows() in Python

Code 4.14 shows an example of using the `csv.writerows()` method to write multiple rows of data to a CSV file in Python. In this example, the `writerows()` method is used to write the header row and the data rows to the CSV file. Each inner list in the list of lists is written as a separate row in the CSV file. The `writerows()` method is useful when you want to write multiple rows of data at once. This operation is produces a CSV file with the same content as shown in Table 4.2.

Exercises

```
1. Create a CSV File
   - Create a CSV file called ‘employees.csv‘ with the following columns:
      - id
      - name
      - department
      - salary
   - Add the following data to the CSV file:
      - 1, John Doe, HR, 50000
      - 2, Jane Smith, IT, 60000
      - 3, Bob Johnson, Finance, 70000
      - 4, Alice Brown, Marketing, 80000
      - 5, Charlie Green, Sales, 90000

2. Read the CSV File using DictReader
   - Read the contents of the CSV file ‘employees.csv‘ using the
     ‘csv.DictReader()‘ function.
   - Print the id, name, department, and salary for each employee in the CSV
     file using the following format:

      Employee ID: 1
      Name: John Doe
      Department: HR
      Salary: 50000

      Employee ID: 2
      Name: Jane Smith
      Department: IT
      Salary: 60000

      ...
```

5

Exception Handling

5.1 Introduction

Exception handling is the process of responding to the occurrence of exceptions in a program. An exception is an event that occurs during the execution of a program that disrupts the normal flow of the program's instructions. Exceptions can occur due to various reasons, such as invalid input, file not found, division by zero, and so on. With exception handling, you can catch and handle exceptions, allowing your program to continue running without crashing. In Python, exception handling is done using the `try`, `except`, and `finally` blocks.

5.2 Try-Except Block

A `try` block is used to wrap the code that may raise an exception. If an exception occurs in the `try` block, the code in the `except` block is executed. The `except` block is used to catch exceptions and handle them. You can specify the type of exception you want to catch, or you can use a generic `except` block to catch all exceptions.

```
try:
    num1 = int(input("Enter a number: "))
    num2 = int(input("Enter another number: "))
    result = num1 / num2
    print("Result:", result)
except ZeroDivisionError:
    print("Error: Division by zero is not allowed.")
except ValueError:
    print("Error: Invalid input. Please enter a valid number.")
except Exception as e:
    print("An unexpected error occurred:", e)
except:
    print("An unexpected error occurred.")
```

Code 5.1: Try-Except Block in Python

Figure 5.1 shows an example of using the `try` and `except` blocks in Python. In this example, the user is prompted to enter two numbers, and the program attempts to divide the first number by the second number. If the user enters a zero for the second number, a `ZeroDivisionError`

exception is raised, and the program prints an error message. If the user enters an invalid input, a `ValueError` exception is raised, and the program prints an error message. If any other unexpected error occurs, the program prints a generic error message. The `except` block can also catch all exceptions using the generic `except` statement. This is useful when you want to handle all types of exceptions in a single block.

5.3 Finally Block

The `finally` block is used to specify a block of code that will be executed regardless of whether an exception occurs or not. The `finally` is often used to perform cleanup actions, such as closing files or releasing resources. The `finally` block is optional and can be used in conjunction with the `try` and `except` blocks.

```
try:
    num1 = int(input("Enter a number: "))
    num2 = int(input("Enter another number: "))
    result = num1 / num2
    print("Result:", result)
except ZeroDivisionError:
    print("Error: Division by zero is not allowed.")
except ValueError:
    print("Error: Invalid input. Please enter a valid number.")
except Exception as e:
    print("An unexpected error occurred:", e)
finally:
    print("I am always executed.")
```

Code 5.2: Finally Block in Python

Figure 5.2 shows an example of using the `finally` block in Python. In this example, the `finally` block is used to print a message indicating that the block is always executed, regardless of whether an exception occurs or not. The `finally` block is executed after the `try` and `except` blocks, and it is useful for performing cleanup actions, such as closing files or releasing resources.

5.4 Types of Exceptions

In Python, there are several built-in exceptions that are raised when certain errors occur. Some common built-in exceptions include:

- `ZeroDivisionError` - Raised when dividing by zero.
- `ValueError` - Raised when a function receives an argument of the correct type but an inappropriate value.
- `TypeError` - Raised when an operation or function is applied to an object of inappropriate type.
- `FileNotFoundError` - Raised when trying to open a file that does not exist.
- `IndexError` - Raised when trying to access an index that is out of range for a list or tuple.
- `KeyError` - Raised when trying to access a dictionary with a key that does not exist.
- `AttributeError` - Raised when an invalid attribute reference is made.
- `ImportError` - Raised when an import statement fails to find the module definition.
- `NameError` - Raised when a local or global name is not found.

- `RuntimeError` - Raised when an error occurs that does not fall into any other category.
- `StopIteration` - Raised when the iterator has no more items to return.
- `KeyboardInterrupt` - Raised when the user interrupts program execution (e.g., by pressing Ctrl+C).

5.5 Raising Exceptions

`raise` statement is used to raise an exception in Python. You can use the `raise` statement to raise a specific exception or to raise a generic exception. The `raise` statement can be used in a `try` block to raise an exception when a certain condition is met. You can also use the `raise` statement to raise a custom exception by creating a new exception class that inherits from the built-in `Exception` class.

```
try:
    num = int(input("Enter a positive number: "))
    if num < 0:
        raise ValueError("Negative number entered.")
    print("You entered:", num)
except ValueError as e:
    print("Error:", e)
```

Code 5.3: Raising Exceptions in Python

Figure 5.3 shows an example of using the `raise` statement to raise an exception in Python. In this example, the user is prompted to enter a positive number. If the user enters a negative number, a `ValueError` exception is raised with a custom error message. The `raise` statement is used to raise the exception, and the `except` block is used to catch the exception and print the error message. The `raise` statement can be used to raise any type of exception, including built-in exceptions and custom exceptions.

5.6 User-Defined Exceptions

In Python, you can create your own custom exceptions by defining a new exception class that inherits from the built-in `Exception` class. A custom exception class can have its own attributes and methods, and it can be used to raise and catch exceptions in your code. Custom exceptions are useful when you want to define specific error conditions that are not covered by the built-in exceptions. You can use custom exceptions to handle specific error conditions in your code and to provide more meaningful error messages.

```
class InvalidAgeException(Exception):
    def __init__(self, message):
        self.message = message
        super().__init__(self.message)

try:
    age = int(input("Enter your age: "))
    if age < 0:
        raise InvalidAgeException("Age cannot be negative.")
    print("Your age is:", age)
```

```
except InvalidAgeException as e:
    print("Error:", e)
except ValueError:
    print("Error: Invalid input. Please enter a valid number.")
```

Code 5.4: User-Defined Exceptions in Python

Figure 5.4 shows an example of creating a custom exception class called `InvalidAgeException` that inherits from the built-in `Exception` class. The custom exception class has an `__init__` method that takes a message as an argument and initializes the exception with the message. In the `try` block, the user is prompted to enter their age. If the user enters a negative age, the `InvalidAgeException` is raised with a custom error message. The `except` block is used to catch the custom exception and print the error message. The `InvalidAgeException` class can be used to handle specific error conditions related to age validation in your code.

Exercises

```
1. Create a Custom Exception Classes
   - Create a custom exception class called ‘InvalidNameException‘ that
     inherits from the built-in ‘Exception‘ class.
   - Create a custom exception class called ‘InvalidDateException‘ that
     inherits from the built-in ‘Exception‘ class.
   - Create a custom exception class called ‘InvalidAgeException‘ that
     inherits from the built-in ‘Exception‘ class.
   - The classes should have an ‘__init__‘ method that takes a message as an
     argument and initializes the exception with the message.

2. Raise the Custom Exception
   - In the ‘try‘ block, prompt the user to enter their name and birthdate in
     the format ‘YYYY-MM-DD‘.
   - If the name is empty or contains numbers, raise the
     ‘InvalidNameException‘ with a custom error message.
   - If the birthdate is not in the correct format, raise the
     ‘InvalidDateException‘ with a custom error message.
   - If the age is less than 0 or greater than 120, raise the
     ‘InvalidAgeException‘ with a custom error message.
   - In the ‘except‘ block, catch the ‘InvalidNameException‘,
     ‘InvalidDateException‘, and ‘InvalidAgeException‘ exceptions and print
     the error message.
   - Also, catch any ‘ValueError‘ exceptions that may occur when converting
     the input to an integer.
   - Finally, print a message indicating that the program has finished
     executing.
```

6

Object-Oriented Programming

6.1 Introduction

6.2 Classes and Objects

6.3 Inheritance

6.4 Polymorphism

6.5 Encapsulation

6.6 Abstraction

7

Modules and Packages

7.1 Introduction

7.2 Creating Modules

7.3 Importing Modules

7.4 Creating Packages

7.5 Importing Packages

8

Regular Expressions

8.1 Introduction

8.2 Match Function

8.3 Search Function

8.4 Findall Function

8.5 Split Function

8.6 Sub Function

9

Database Connectivity

9.1 Introduction

9.2 Connecting to a Database

9.3 Creating a Table

9.4 Inserting Data

9.5 Selecting Data

9.6 Updating Data

9.7 Deleting Data

10

Graphical User Interface

11

References

A. Books

-

B. Other Sources

-