Synthesis Lectures on Computer Science

SYNTHESIS
COLLECTION OF TECHNOLOGY

Paul A. Gagniuc

Coding Examples from Simple to Complex

Applications in Python™

Springer

Synthesis Lectures on Computer Science

The series publishes short books on general computer science topics that will appeal to advanced students, researchers, and practitioners in a variety of areas within computer science.

Paul A. Gagniuc

Coding Examples from Simple to Complex

Applications in Python™

Springer

Paul A. Gagniuc
Department of Engineering in Foreign
Languages, Faculty of Engineering in Foreign
Languages
National University of Science and Technology
Politehnica Bucharest
Bucharest, Romania

ISSN 1932-1228 ISSN 1932-1686 (electronic)
Synthesis Lectures on Computer Science
ISBN 978-3-031-53811-7 ISBN 978-3-031-53812-4 (eBook)
https://doi.org/10.1007/978-3-031-53812-4

This Springer imprint is published by the registered company Springer Nature Switzerland AG
The registered company address is: Gewerbestrasse 11, 6330 Cham, Switzerland

Paper in this product is recyclable.

Foreword

The book *Coding Examples from Simple to Complex—Applications in Python*™ by Paul Aurelian Gagniuc is a very hands-on introduction to programming in Python, appealing to readers ranging from novices making their first steps in the universe of programming to more seasoned developers, that can use a very rich reference of code examples. Since this is the main feature of this work, teaching through examples, over 200, each chapter exemplifying the key concepts by exercises which are implemented, commented, and explained in great detail.

The chosen language is Python, most probably one of the most popular programming languages nowadays. While most learning institutions introduce programming languages using the Python language, readers will benefit the wealth of the examples in this book, as well as from being exposed to more advanced programming techniques.

The structure is well-thought, starting with traditional starting points in variable declaration, expressions, control statements, lists (arrays), and functions and continuing with objects and advanced techniques. The author focuses on imperative programming techniques, more suitable for beginners, however, treating also functional programming and object-oriented programming in the respective chapters. The examples support the chapters in a logical succession, one advantage being that a simplified solution is shown before an optimized one, useful for a deeper understanding of the problem.

The book continues with the moderate examples section, in which more real-world usages are shown, ranging from topics such as string manipulation, more advanced matrix operations, sorting algorithms, bitwise operations and encodings, and statistics. As examples are implemented without the use of other libraries except the standard library, they are of great teaching value, in helping practitioners truly understand the inner workings of concepts.

Where the book is of interest to more advanced developers or researchers in different fields is in the complex examples section, covering novel, state-of-the-art algorithms such

as spectral forest or complex usage of Markov Chains, an area in which the author is a renowned expert.

Andrei Vasilateanu
Friend and Vice Dean
Faculty of Engineering in Foreign
Languages
National University of Science
and Technology Politehnica Bucharest
Bucharest, Romania

Preface

Python is highly regarded for its simplicity, readability, and versatility, making it a popular choice for server-side (back-end) web development, data analysis, artificial intelligence, and scientific computing. This book presents a comprehensive guide to Python™, offering an exploration from fundamental concepts to advanced programming techniques. This work is designed for readers ranging from beginners to experienced developers and scientists, aiming to equip them with a thorough understanding of Python's capabilities.

Key Features of the Book

Hands-on learning. Over 200 practical examples are included, in order to reinforce the understanding of Python concepts and computer programming principles.

Comprehensive coverage. The book also covers Python essentials such as variables, conditionals, loops, lists, functions, and JSON handling, providing a solid foundation for beginners.

Advanced techniques. Readers will explore advanced topics such as matrix operations, recursion, object-oriented programming, and more, enhancing their programming skills.

Real-world applications. Practical applications of Python are demonstrated, including data manipulation, graphical interfaces, and file operations, showing how Python can be applied in various scenarios.

Mathematical implementation. The book guides on the implementation of mathematical formulas and concepts in Python code, beneficial for scientific computing.

This comprehensive exploration of Python is tailored for a wide range of learners and professionals. It systematically unfolds Python programming, showcasing its systematic and rigorous capabilities in the field of software engineering. The journey through this book will enhance the proficiency of the readers in Python, enabling them to effectively utilize the language in various projects and applications. This book is part of a series of

book titles that aims to mirror these examples and their explanations, as close to each other as possible. Thus, these examples can also be found in other computer languages.

Bucharest, Romania Paul A. Gagniuc

Contents

Introduction

1

Python, a high-level, interpreted computer language, has a history that dates back to the late 1980s [1]. It was conceived by *Guido van Rossum* at *Centrum Wiskunde & Informatica* (CWI) in the Netherlands as a successor to the ABC language, which was itself designed for teaching and prototyping. *Van Rossum* started implementing Python during the 1989 Christmas holidays, with a goal to create a language that emphasized code readability and simplicity [2]. Python version 0.9.0, released in 1991, introduced several features still central to Python, including classes with inheritance, exception handling, and functions. The name Python is inspired by the British comedy group *Monty Python*, reflecting the goal of *Van Rossum* to make programming fun and accessible. Throughout the 1990s, the Python scripting language continued to evolve, with multiple contributors adding to its development. The release of Python 1.0 in 1994 included functional programming tools like lambda, map, filter, and reduce. Python 2.0, released in 2000, marked a significant milestone, introducing list comprehensions and a garbage collection system capable of collecting reference cycles. It also solidified the community-led development model, with the formation of *Python Enhancement Proposals* (PEPs), a mechanism for proposing major new features. Python 3.0, released in 2008, was a major, backward-incompatible release, which was not initially widely adopted due to its incompatibility with Python 2.x. However, it introduced several improvements to the language, including a more consistent and clean syntax, better Unicode support, and changes to the standard library. Over time, Python 3 gained traction, especially as the Python community transitioned away from Python 2, which saw its final release, 2.7, in 2010. Today, Python stands as one of the most popular computer programming languages, widely used in web development, data analysis, artificial intelligence, scientific computing, and more [3]. Its popularity is attributed to its readability, ease of learning, and the vast ecosystem of libraries [4, 5].

P. A. Gagniuc, *Coding Examples from Simple to Complex*, Synthesis Lectures on Computer Science, https://doi.org/10.1007/978-3-031-53812-4_1

1.1 Future of Python

The trajectory of Python appears increasingly influential and integral in various technology sectors [6, 7]. Its growing popularity in emerging fields such as data science, machine learning, and artificial intelligence positions it as a critical tool for future innovations [8, 9]. The simplicity of the language and readability make it an ideal first language for beginners in programming, suggesting its role in education will continue to expand [10]. Furthermore, the large and active community is a vital asset, driving the evolution of the language to meet new challenges and requirements. Future enhancements of Python are likely to focus on performance optimization and concurrency, areas where it has faced criticism compared to languages like Java and C++. Efforts such as the *PyPy* project, which aims to increase the execution speed of Python, and ongoing improvements in asynchronous programming capabilities, are steps in addressing these challenges. As hardware capabilities grow, the ability of Python to interface with low-level languages and use these resources will be crucial. Another area of focus will be the continued development of the rich ecosystem of Python libraries and frameworks, which are instrumental in its widespread adoption. Libraries such as *NumPy*, *pandas*, *TensorFlow*, and *PyTorch* are central to the dominance of Python in scientific computing and machine learning, and their ongoing development will further solidify this position [11–15]. The role of Python in web development, although less prominent than languages like JavaScript, is also set to expand with frameworks like *Django* and *Flask*, which simplify the process of developing complex web applications [16]. The future of Python is closely tied to its adaptability, the strength of its community, and its ability to stay relevant in a rapidly changing technological landscape. With its foundation in simplicity and readability, combined with powerful capabilities through its libraries, Python is well-positioned to remain a key player in the programming world for years to come.

1.2 The Content is Native

This work showcases native Python implementations from basic to complex, and is addressed to a large audience, from beginners to Ph.D. students and even mature scientists and engineers. The first part of this book describes the use of variables, conditional branching and loops. Variables, as foundational elements of programming languages, form the focus of the first chapter. Topics covered include variable declaration and initialization, nomenclature conventions, and the composition of a basic Python program. Additionally, discussions will encompass assignment, variable types, fundamental arithmetic operations, and related subjects. Also, conditional branching mechanisms, which facilitate decision-making processes and the execution of divergent code segments based on predetermined conditions, are explored in detail. Emphasis is placed on a variety of conditional statements such as “if-then,” “if-then-else,” and “if-then-elseif-else.” These constructs enable

the manipulation of program flow and responsiveness to varying scenarios. Next, the concept of loops is explored in detail, as it is instrumental in iteratively executing code blocks and enhancing program efficiency. A comprehensive exploration of both "While" and "For" loops is undertaken. Topics of interest include count-controlled loops, array traversal, and intricate mathematical computations. In a second part of the book, more complex variables such as arrays (or lists) are described by example. The subject of multidimensional traversal of arrays is also covered, and then some matrix operations are shown. Arrays (called lists in Python), as fundamental data structures for organizing and manipulating data collections, are scrutinized in a dedicated chapter. Topics encompass basic array operations such as element addition and retrieval, length calculation, and array traversal. The employment of various loop types for array traversal is discussed in detail. Moreover, the traversal and manipulation of these multidimensional arrays are explored comprehensively. The discussion extends to encompass 2D and 3D arrays, matrix operations, and transformations including transposition and rotation. Furthermore, matrix operations are shown as pivotal in mathematical and graphical contexts by using specific examples. Subjects addressed include summation, multiplication, diagonal extraction, transposition, and related matrix operations. In a third part of this paper, functions, object constructors, and methods are thoroughly explored from several angles, and the JSON method is presented as an exchange medium between different data formats. Functions, instrumental in code reusability and modularity, are the primary focus of an extensive chapter. Both built-in and user-defined functions are explored in depth. Topics encompass function creation, parameterization, and return value handling. Additionally, discussions extend to recursion, logical operations, sorting algorithms, statistical computations, and diverse practical examples showcasing the versatility of functions in Python. The conceptualization and implementation of objects, their properties, and methods are expounded upon in a separate chapter. Object constructors, object instantiation, and the inclusion of methods within objects are thoroughly explored. Practical examples underscore the principles of object-oriented programming in Python. Also, JSON (JavaScript Object Notation) as a prevalent data interchange format is the central theme of this chapter. The chapter addresses the conversion between Python objects and JSON, manipulation of JSON data, and the handling of complex JSON structures. In the fourth part of the book, the reader encounters moderate and complex examples and, most importantly, cases related to randomness and programming. The chapter on moderate and complex example serves as a culmination of Python knowledge, presenting intricate examples that demonstrate the utility of the language in solving real-world problems. Topics include statistical analysis, sequence alignment, and text processing, offering insights into advanced programming techniques. Also, the chapter on randomness discusses methods that show how to model a random process. In the last part of the book, the discussion focuses on Python applications that are more language specific. The chapter explores some Python features, encompassing *base64* encoding and decoding using built-in functions and file I/O mechanisms. Among

others, the final chapter also introduces readers to graphics programming in Python, covering the creation of visual elements, and interactive graphics using libraries such as *tinker*.

Variables

2

A variable can be conceptualized as a symbolic representation or an abstract entity that holds information [1]. This information can take various forms, from simple numerical values, strings of text, to more complex data structures. The central essence of a variable lies in its ability to change or vary, making it indispensable in algorithms and computational processes [1]. Variables are foundational to computer programming because they allow for the storage and manipulation of data. Each variable has an associated data type, which dictates the kind of information the variable can store. For instance, an integer data type variable can store whole numbers, whereas a floating-point data type might store numbers with decimal points. When a variable is declared in a computer program, a specific portion of the computer memory is allocated to store its value. This allocation ensures that when the value of the variable is called upon or modified, the program knows exactly where to look in memory. Each variable has a unique memory address, which acts like a reference point for any computational operation involving that variable. Variables also possess attributes such as scope (determining where in a program a variable can be accessed) and lifetime (indicating how long the variable remains in memory). The importance of these attributes becomes evident in more advanced programming tasks, such as managing memory or optimizing code for performance. In scientific computing, variables often represent physical quantities or abstract mathematical constructs. Their ability to change values dynamically allows for the simulation of real-world systems, from modeling the motion of celestial bodies to predicting weather patterns. Scientists can run multiple scenarios or simulations to analyze different outcomes and derive meaningful conclusions just by adjusting the values of these variables. Thus, variables serve as the backbone of computational algorithms and programs. Their dynamic nature, combined with the precise control they offer over data manipulation, makes them a cornerstone in

P. A. Gagniuc, *Coding Examples from Simple to Complex*, Synthesis Lectures
on Computer Science, https://doi.org/10.1007/978-3-031-53812-4_2

the world of computer science and scientific computing. Thus, the examples shown below start from basic exercises that familiarize the reader with the notion of variables.

2.1.1 Ex. (1) – Commenting inside code

```
# this is a comment in Python

"""
This is a multi-line
comment in Python.
It can span several lines.
"""
```

Output:

In Python, the hash "#" character is used to denote a single-line comment. Anything that follows the hsdh "#" on that same line is treated as a comment and will not be executed or interpreted as code by the Python engine. Instead, it is meant to provide context or explanations for developers reading the code. Also, multi-line comments or block comments are typically created using triple quotes, either triple single quotes (`'''`) or triple double quotes (`"""`).

2.1.2 Ex. (2) – Naming variables

```
A = 1
a = 2
a1 = 3
a_1 = 4

print(A)
print(a)
print(a1)
print(a_1)
```

Output:

```
1
2
3
4
```

This code initializes four variables with distinct names and values. The variable *A* is assigned the value 1, while the variable *a* is assigned the value 2. Similarly, *a1* is given the value 3, and *a_1* is assigned the value 4. Following these assignments, the values of these variables are printed out sequentially using the *print* function. First, the value of *A* is printed, followed by the values of *a*, *a1*, and finally *a_1*. It is worth noting that Python is case-sensitive, so the variable *A* is different from the variable *a*.

2.1.3 Ex. (3) – Write your first Python program

```
a = 3
b = 5
c = a + b
print(c)
```

Output:

```
8
```

The given Python code from above initializes a variable *a* with a value of 3 and a variable *b* with a value of 5. It then calculates the sum of these two variables and assigns the result to a third variable named *c*. Next, the value of *c* is printed to the console or displayed using a function named *print*. The output of this code is 8.

2.1.4 Ex. (4) – The meaning of "a = b"

```
a = 3
b = a
print(b)
```

Output:

```
3
```

This Python code begins by assigning the value 3 to the variable *a*. Following that, the value of *a* (which is 3) is assigned to another variable named *b*. Next, the *print*(*b*) statement outputs the value of *b*, which would display 3. Therefore, this source code points out the passing (reassignment) of values from one variable to another.

2.1.5 Ex. (5) – Assign and reassign with type change

```
# In Python, one simply assigns a
# value to a variable to declare it.

# Assigns a string 'text'
# to variable a.
a = 'text';

# a = 0; # For sport, one can
# uncomment this line (a = 0;)
# to reassign a to a numeric value.

# Assigns a string 'text'
# to variable b.
b = 'text';
b = 0 # Reassigns b to a numeric value.
```

Output:

In Python, there is no special keyword for declaring a variable, and the type of a variable is automatically determined by the value assigned to it. Thus, Python is dynamically typed, and the user can change the value and the type of a variable at any time. This nice feature is now present in many computer languages and is perhaps one of the most useful additions to modern computer programming because it allows software developers to focus on method rather than syntax. Therefore, unlike the old times, Python will not raise any errors when the user changes the type of a variable by simply assigning a new value of a different type.

2.1.6 Ex. (6) – Basic mathematical operations

```
a = 3
b = 2
c = a + b / 2 - a * b
print(c)
```

Output:

```
-2
```

The above Python code first assigns the value 3 to the variable *a* and the value 2 to the variable *b*. Next, it performs a series of arithmetic operations using these two variables. Specifically, it divides *b* by 2, then adds the result to *a*, and from that sum, it subtracts the product of *a* multiplied by *b*. The final result of these calculations is assigned to the variable *c*. Lastly, the value of *c* is printed out to the console.

2.1.7 Ex. (7) – The meaning of *modulo* operator

```
a = 3
a = a % 2
print(a)
```

Output:

```
1
```

The code starts by assigning the value 3 to the variable *a*. Next, it modifies the value of *a* by setting it to the remainder when a is divided by 2, which is done using the modulus (%) operator. The modulus operation determines the remainder of the division of *a* by 2. Thus, 3 divided by 2 gives a quotient of 1 and a remainder of 1. Therefore, after the modulus operation, the value of *a* becomes 1. Next, it uses a *print*(*a*) statement to display the value of *a* to the console window.

2.1.8 Ex. (8) – The meaning of "a = a + 1"

```
a = 2
a = a + 1
print(a)
```

Output:

```
3
```

The given Python code starts by assigning the value 2 to the variable *a*. It then increments the value of *a* by 1. Next, it prints the value of *a*, which would now be 3. Therefore, the output shows the value 3 in the console window.

2.1.9 Ex. (9) – The agregate asignment

```
a = 2
a += 1
print(a)
```

Output:

```
3
```

The given Python code first assigns the value 2 to the variable *a*. Then, it increments the value of *a* by 1 using the "+=" operator, which is shorthand for $a = a + 1$. After these operations, the value of *a* becomes 3. Next, the code prints the value of *a* to the console window by using the *print*(*a*) statement.

2.1.10 Ex. (10) – The plus operator

```
a = 2;
a = a + a;
print(a);
```

Output:

```
1
```

The given Python code first assigns the value 2 to the variable *a*. Then, it increases the value of *a* by *a* using the "+" operator. Namely, the future value of *a* is the current value of *a* plus the current value of *a*, which is shorthand for $a = a + a$. After these operations, the value of *a* becomes 4. Next, it prints *a* using the *print*(*a*) statement.

2.1.11 Ex. (11) – The minus operator

```
a = 2
a = a - a
a = a - 1
print(a)
```

Output:

```
-1
```

This code is complementary to the previous example and it starts by assigning value 2 to variable *a*. Next, it decreases the value of *a* by using the "−" operator. Namely, the future value of *a* will be the current value of *a* minus the current value of *a*, which is short for $a = a - a$. On the next line, for the sake of code variability, the variable *a* is further reduced by 1. Next, the *print*(*a*) statement displays the value of *a*, which is -1.

2.1.12 Ex. (12) – Variable playground

```
a = 2
a += a - 1
a += a - 1
print(a)
```

Output:

```
5
```

The code from above performs a series of operations on the variable *a*. Initially, *a* is assigned a value of 2. The code then performs a series of operations on *a*. First, *a* is incremented by $a - 1$. Since *a* is 2 at this point, $a - 1$ equals 1, thus, *a* becomes $2 + 1$, which is 3. The next line repeats a similar operation: *a* is incremented by $a - 1$ again. Now, *a* is 3, so $a - 1$ equals 2, and thus *a* becomes $3 + 2$, which is 5. Next, the *print*(*a*) statement outputs the value of *a*, which is 5. Thus, this simple code snippet demonstrates how variables can be modified and updated in Python, reflecting the dynamic nature of variable assignments and arithmetic operations.

2.1.13 Ex. (13) – Swap values

```
a = 3
b = 7
t = 0

t = a
a = b
b = t

print('a =', a)
print('b =', b)
```

Output:

```
a = 7
b = 3
```

The given code initializes three variables: *a*, *b*, and *t*, with the values 3, 7, and 0 respectively. The purpose of the code is to swap the values of *a* and *b* without using any direct arithmetic operations or additional variables. To achieve this, the value of *a* is first stored in the temporary variable *t*. Then, the value of *b* is assigned to *a*, effectively overwriting *a* original value. Lastly, the value stored in *t* (which is the original value of *a*) is assigned to *b*, completing the swap. After the swapping operation, two *print* statements display the updated values of *a* and *b*, showing that their values have indeed been exchanged. Thus, after the code executes, the output will be "a = 7" and "b = 3".

2.1.14 Ex. (14) – Empty a variable

```
a = 3
b = a + 7
a = None
print(a)
print(b)
```

```
Output:

None
10
```

The Python code from above initializes a variable *a* with the value 3. Then, it initializes another variable *b* and assigns it the result of adding *a* to 7, making the value of *b* equal to 10. Afterward, the value of *a* is set to *None*. Next, the code prints the value of *a*, which is *None*, and then prints the value of *b*, which remains 10.

2.1.15 Ex. (15) – Line continuation

```
s = (1 + 2 + 3 +
     4 + 5 + 6 + 7 + 8)
print(s)
```

```
Output:

36
```

The given code is performing an arithmetic operation where multiple numbers are being added together. It starts by adding the numbers 1, 2, and 3. The addition then continues on the next line with the numbers 4 through 8. After computing the sum, which is stored in the variable *s*, the result is printed to the console window by using the *print*(*s*) statement.

2.1.16 Ex. (16) – Formated output

```
a = 3
b = 7
c = 10
r = "a = " + str(a) + " and b = " + str(b)
t = " is a number.\n"
l = str((a + b / c)) + t
print(l + r)
```

Output:

```
3.7 is a number.
a = 3 and b = 7
```

In this code snippet, several variables are declared and manipulated. First, variables *a*, *b*, and *c* are declared and initialized with numerical values 3, 7, and 10, respectively. Next, a string variable *r* is created and assigned a value that concatenates the string "a =" with the value of *a*, then "and b =" with the value of *b*. This will create a string that describes the values of variables *a* and *b*. Another string variable *t* is initialized with the string "is a number.\n", where "\n" is an escape character for a new line. The variable *l* is then created, and it stores the result of an arithmetic operation that adds *a* to the division of *b* by *c*. This result is then concatenated with the string stored in *t*. Next, the *print* function is called to display the combined value of *l* and *r*.

Conditional Branching

3

Conditional branching, also colloquially known as decision-making in source code, is a fundamental concept in computer science and algorithm design, enabling systems to perform different computations depending on whether a specified boolean condition evaluates to true or false [1]. At its core, conditional branching simulates the logical reasoning humans naturally employ in decision-making processes. For instance, if it is raining, one might choose to take an umbrella. Similarly, in code, such decisions are represented using constructs like if-else statements. In structured programming languages, these branches are often encapsulated within if, else if, and else constructs. For example, an algorithm that determines whether a number is positive, negative, or zero might use conditional branching to evaluate the number and return an appropriate response. Moreover, conditional branching extends beyond simple binary choices. The switch-case construct, present in many programming languages, allows for multiple conditions to be evaluated in sequence, facilitating decisions amongst numerous potential pathways. However, in Python the notion of switch does not exist, but, it can be simulated. From a computational efficiency perspective, conditional branches introduce the concept of non-linear code execution. Rather than executing a sequence of instructions linearly, the program may skip over large chunks of code based on the outcome of a condition. This allows for more efficient code execution but introduces complexity in terms of ensuring each branch leads to a valid and expected program state. In modern architectures, however, excessive branching can be detrimental to performance due to the mechanics of pipelining and branch prediction in CPUs. A mispredicted branch can result in a CPU pipeline stall, leading to wasted clock cycles. As a result, while conditional branching is a very powerful tool, understanding its implications on underlying hardware is crucial for performance-critical applications. Thus, decision-making is indispensable in software

P. A. Gagniuc, *Coding Examples from Simple to Complex*, Synthesis Lectures on Computer Science, https://doi.org/10.1007/978-3-031-53812-4_3

design, enabling dynamic decision-making based on evolving conditions. Its effective use, combined with an understanding of its impact on computational efficiency, remains paramount for software development.

3.1.1 Ex. (17) – *If then else* – conditional statements (I)

```
a = 4
b = 7
if a < b:
    print(a)
else:
    print(b)
```

Output:

```
4
```

This Python code defines two variables, a with a value of 4 and b with a value of 7. It then checks if the value of a is less than the value of b using an *if-else* statement. If the condition is true, meaning if a is indeed less than b, it will print the value of a to the console window. If the condition is false, it will print the value of b. In this case, since 4 is less than 7, it will print the value of a, which is 4.

3.1.2 Ex. (18) – *If then else* – conditional statements (II)

```
a = 2
b = 3
c = 1

if a < b:
    c = 0

print(c)
```

Output:

```
0
```

This Python code initializes three variables: a is assigned the value 2, b is assigned the value 3, and c is assigned the value 1. It then checks if the value of a is less than the value of b. If this condition is true, which it is in this case since 2 is less than 3, the value of c is updated to 0. The value of c is printed. Given the initial values and the condition provided, the output will be 0.

3.1.3 Ex. (19) – *If then else* - conditional statements (III)

```
a = 1
b = 2
c = 3

if a < b:
    c += 1
else:
    c -= 1

print("c=" + str(c))
```

Output:

```
c=4
```

The given Python source code from above initializes three variables: a is assigned a value of 1, b is assigned a value of 2, and c is assigned a value of 3. Then, there is an *if-else* conditional statement that checks if the value of a is less than the value of b. If this condition is true, the value of c is incremented by 1. If the condition is not true (i.e., if a is not less than b), the value of c is decremented by 1. After evaluating this conditional statement, the code prints to the console the value of c, with a prefix "c=". After the *if-else* statement, the value of c would be 4 because the condition $a < b$ (1 is less than 2) is true. Note that the content of c is converted into a string by using the built-in function *str*.

3.1.4 Ex. (20) – *If then elseif else* - conditional statments

```
a = 1
b = 2
c = 3

if a < b:
    c -= 1
elif b == c:
    c += 1
else:
    c = 0

print("c=" + str(c))
```

Output:

```
c=2
```

The above Python code initializes three variables: a with a value of 1, b with a value of 2, and c with a value of 3. The code then contains a conditional structure to manipulate the value of c based on certain conditions. If the value of a is less than b, then 1 is subtracted from the current value of c. If that condition is not met but the value of b is equal to the value of c, then 1 is added to the current value of c. If neither of these conditions is satisfied, the value of c is set to 0. After evaluating these conditions, the code prints the string "c=" followed by the current value of c.

3.1.5 Ex. (21) – Simulation of the *Switch* statement

```
a = 1
b = 0

if a == 0:
    b = 11
elif a == 1:
    b = 64
elif a == 2:
    b = 33

print(b)
```

Output:

```
64
```

In Python, there is no direct equivalent of the *switch* statement found in languages such as JavaScript. Instead, one can use an *if-elif-else* structure to achieve similar functionality. This Python source code starts by declaring two variables, a and b, and assigning them the values of 1 and 0 respectively. Then, a switch statement is used to evaluate the value of a. If a is 0, the value 11 will be assigned to b. If a is 1, then b will be assigned the value 64. If a is 2, the value 33 will be assigned to b. Lastly, the value of b is printed out. Given the initial value of a is 1, the printed value of b will be 64.

Loops

4

For imperative computer programming languages, loops hold a paramount position [1]. They are fundamental structures that facilitate the repeated execution of a set of instructions, enabling efficient automation and repetitive task handling. From a conceptual viewpoint, a loop is a mechanism by which a process can be reiterated until a specific condition or set of conditions is met. This cyclic execution allows for the efficient handling of tasks that follow a recurrent nature. There are several types of loops, primarily distinguished by their control mechanisms: (a) For-Loop, that is generally used when the number of iterations is known in advance. The loop contains an initializer, a condition, and an iterator. It commences with the initialization, checks the condition, and post-execution, the iterator modifies the loop variable, leading to the next iteration or exit. (b) While-Loop, that is predominantly used when the number of iterations is not predetermined, the while loop checks a condition before every iteration. If the condition evaluates to true, the loop body is executed. Each iteration within a loop is commonly referred to as a "cycle". In every cycle, the computational instructions are reevaluated, often with altered variables, leading to different outcomes in each iteration. It is vital to ensure that loops have a definitive termination point or a condition that will be met, to prevent infinite looping, which can lead to system hang-ups or overconsumption of computational resources. Loops are foundational constructs in computer programming that harness the power of computation by enabling repetitive execution based on conditions. Some basic examples are shown here in order to familiarize the reader with these structures.

P. A. Gagniuc, *Coding Examples from Simple to Complex*, Synthesis Lectures
on Computer Science, https://doi.org/10.1007/978-3-031-53812-4_4

4.1.1 Ex. (22) – *While loop*

```
i = 0

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

Output:

```
i = 0
i = 1
i = 2
i = 3
i = 4
```

The code initializes a variable named i with the value of 0. Then, a while loop starts, which continues executing its body as long as the condition $i < 5$ holds true. Inside the loop, a *print* function is called, which displays the current value of i in the format "i = [value of i]". After printing the value, the value of i is incremented by 1 using the $i += 1$ statement. As a result, the loop will print the values of i from 0 to 4. Once i reaches 5, the condition $i < 5$ will no longer be true, and the loop will terminate.

4.1.2 Ex. (23) – *Do while*

```
i = 0

while True:
    print("i = " + str(i))
    i += 1
    if i >= 5:
        break
```

Output:

```
i = 0
i = 1
i = 2
i = 3
i = 4
```

This Python code initializes a variable i with a value of 0. It then enters a special simulation of "do-while" loop (as Python does not have a "do-loop" structure). Inside the loop, a *print* function is called to display the current value of i, which is concatenated with the string "i=". Notice that *str*(i) function is used to convert the numeric value of i to a string so it can be concatenated with another string. Next, the value of i is incremented by 1. The loop continues to execute as long as the value of i is less than 5. Once i reaches 5, the loop stops. Thus, the output of this code would be a sequence of printed statements displaying "i = 0", "i = 1", "i = 2", "i = 3", and "i = 4". In other words, the *while true* loop creates an infinite loop, which is only exited when the *if* condition inside the loop fails ($i < 5$) and break is executed. Note that this Python code mimics the behavior of a *do-while* loop where the loop body is guaranteed to execute at least once.

4.1.3 Ex. (24) – Simple *for loop*

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

Output:

```
0 1 2 3 4
```

The code above is a *for-loop* that initializes a variable *i* to 0. It then checks if the value of *i* is less than 5. If the condition is true, the code inside the loop is executed. Inside the loop, there is a function call to *print*(*i*), which would display the value of *i*. After executing the loop body, *i* is incremented by 1. The loop will continue to execute as long as *i* is less than 5. As a result, the numbers 0 through 4 will be passed to the *print* function one at a time.

4.1.4 Ex. (25) – Revers by substraction from the upper limit

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

Output:

```
5 4 3 2 1
```

The given code uses a loop with a variable *i* set to 0. The loop continues to run as long as the value of *i* is less than 5. With each iteration of the loop, the value of *i* increases by 1. Inside the loop, there is a function call to *print()* which takes the expression 5-*i* as its argument. This means that for each iteration of the loop, the function will print a value that starts from 5 (when *i* is 0) and decrements by 1 with each subsequent iteration. Thus, the sequence of numbers printed will be 5, 4, 3, 2, and finally 1.

4.1.5 Ex. (26) – Reverse *for-loop*

```
for i in range(10, 5, -1):
    print(i)
```

Output:

```
10 9 8 7 6
```

The given Python code initializes a *for-loop* with the variable *i* set to 10. The loop continues executing as long as the value of *i* is greater than 5. With each iteration of the loop, the value of *i* is decremented by 1. Inside the loop body, there is a *print*(*i*) statement, which would ideally print the current value of *i*.

4.1.6 Ex. (27) – The meaning of before and after

```
i = 10
while i > 5:
    i -= 1
    print(i)
    i -= 1
```

Output:

```
9 7 5
```

The Python from above uses a *while-loop* to show the result of operations on a variable, before and after the content of the variable is printed. The loop starts with *i* at 10, and in each iteration, *i* is decremented twice: once before the *print* statement and once at the end of the loop body. The loop continues as long as *i* is greater than 5.

4.1.7 Ex. (28) – Revers by substraction from the upper limit variable

```
a = 5

for i in range(a):
    print(a - i)
```

Output:

```
5 4 3 2 1
```

The provided Python code initializes a variable a with the value of 5. It then uses a *for-loop* to iterate from 0 up to, but not including, the value of *a* (which is 5). During each iteration of the loop, it calls a function named *print* with the argument *a-i*. This means that with each iteration, it will print the result of subtracting the current loop index *i* from *a*. In this specific case, the sequence of numbers that will be printed is 5, 4, 3, 2, and 1.

4.1.8 Ex. (29) – *For loop* sumation

```
a = 0

for i in range(1, 6):
    a = a + (i + 4 * 3)

print(a)
```

Output:

```
75
```

The above code initializes a variable *a* with a value of 0. Following this initialization, there is a *for-loop* that runs 5 times (remember that in Python, the upper limit is non-inclusive). Within each iteration of the loop, the value of *i* (which starts from 1 and increments by 1 each time) is added to the product of 4 multiplied by 3. The result of this addition is then added to the current value of *a*. Essentially, during each loop iteration, 12 (which is 4 multiplied by 3) is added to the value of *i* and the sum is added to *a*. After the loop completes its 5 iterations, the *print* function is called to display the final value of *a*. The *print* function here outputs the final value of *a* to the console.

4.1.9 Ex. (30) – Simple counter sumation

```
a = 0

for i in range(11):
    a = a + i

print(a)
```

Output:

```
55
```

The given Python code initializes a variable named *a* with a value of 0. Then, there is a *for-loop* that iterates 11 times, starting with the value 0 up to, but not including, the value 11. Within each iteration of the loop, the value of *i* (which is the loop counter) is added to the current value of *a*. As a result, *a* accumulates the sum of integers from 0 to 10. After the loop completes its execution, the value of *a* (which will be the sum of the integers from 0 to 10) is printed out.

4.1.10 Ex. (31) – Sum all results of addition of 1 in a $n \times n$ cycle

```
r = 0
for i in range(10):
    for j in range(10):
        r += 1

print(r)
```

Output:

```
100
```

The Python code initializes a variable *r* with the value 0. It then sets up a nested loop structure: the outer loop runs with the variable *i* from 0 up to, but not including, 10, and for each iteration of this outer loop, an inner loop runs with the variable *j* also from 0 up to, but not including, 10. During each iteration of the inner loop, the value of *r* is incremented by 1. As a result, the inner loop runs a total of 100 times (10 times for each of the 10 iterations of the outer loop). Hence, by the end of the nested loops, the value of *r* will be 100. After the loops are finished, the code prints the value of *r*, which will display the number 100.

4.1.11 Ex. (32) – Sum all results of addition of 3 in a $n \times n$ cycle

```
r = 0
for i in range(4):
    for j in range(4):
        r += 3

print(r)
```

Output:

```
48
```

The given Python code initializes a variable *r* with the value of 0. It then uses a nested for-loop, where the outer loop runs 4 times (with the loop variable *i* ranging from 0 to

3) and the inner loop also runs 4 times (with the loop variable *j* ranging from 0 to 3). For each iteration of the inner loop, the value of *r* is incremented by 3. Since there are a total of 16 iterations (4 from the outer loop multiplied by 4 from the inner loop), *r* is incremented by 3 a total of 16 times. As a result, by the end of these nested loops, the value of *r* becomes 48. Next, the code uses the *print* function to display the value of *r*, which would output the number 48.

4.1.12 Ex. (33) – Sum all results of the multiplication between *i* and *j*

```
r = 0
for i in range(10):
    for j in range(10):
        r += j * i

print(r)
```

Output:

```
2025
```

The Python code initializes a variable *r* with a value of 0. It then has a nested loop where the outer loop uses a variable *i* which runs from 0 to 9, and the inner loop uses a variable *j* which also runs from 0 to 9 (again, please remember that in Python, the upper limit is non-inclusive). For each combination of *i* and *j*, the product of *i* and *j* is calculated and added to the value of *r*. After both loops have completed their iterations, the accumulated total in *r* is then printed out. Thus, the code computes the sum of products of all possible combinations of *i* and *j* within the range specified.

4.1.13 Ex. (34) – Nested for loops and summation of counter variables

```
a = 0
m = 3
n = 5

for j in range(1, m + 1):
    for i in range(1, n + 1):
        a = a + (i + j * 3)

print(a)
```

Output:

```
135
```

The Python code initializes three variables: *a* with a value of 0, *m* with a value of 3, and n with a value of 5. It then contains a nested loop where the outer loop runs as long as *j* is less than or equal to *m* (3 times in this case), and for each iteration of this outer loop, an inner loop runs as long as *i* is less than or equal to *n* (5 times). Within the inner loop, the value of *a* is incremented by the sum of *i* and three times *j*. After both loops have finished executing, the accumulated value of *a* is printed out. The purpose of this code is to accumulate a sum based on the conditions and limits set by the variables *m* and *n*.

4.1.14 Ex. (35) – Nested for loops and summation based on the inner counter

```
a = 0

# In Python, range
# end is exclusive.

for j in range(1, 4):
    for i in range(1, 6):
        a = a + (i + 1 * 3)

print(a)
```

Output:

```
90
```

The given Python code initializes a variable a with the value of 0. It then uses a nested for-loop structure to iterate and modify the value of a. The outer loop, controlled by the variable j, runs three times, as j goes from 1 through 3 inclusive. Inside this outer loop, there is an inner loop controlled by the variable i, which runs five times for each iteration of the outer loop, since i goes from 1 through 5 inclusive. For every iteration of the inner loop, the value of a is incremented by the result of the expression (i + 1*3). This expression adds i to the product of 1 and 3. Given the rules of arithmetic operation precedence, the multiplication is performed before the addition, thus, the expression is equivalent to (i + 3). Given the loop structures, this means the operation (i + 3) is executed a total of 15 times (3 times for the outer loop multiplied by 5 times for the inner loop). Next, after both loops have completed their iterations, the value of a is printed out.

4.1.15 Ex. (36) – Nested for loops & summation based on counters and upper limits (I)

```
a = 0
m = 3
n = 5

for j in range(1, m + 1):
    for i in range(1, n + 1):
        a = a + (i + j * m)

print(a)
```

Output:

```
135
```

The given Python code initializes three variables: a with a value of 0, m with a value of 3, and n with a value of 5. The code then sets up a nested loop structure with an outer loop running from 1 through the value of m (inclusive) and an inner loop running from 1 through the value of n (inclusive). Inside the innermost part of this nested loop, the value of a is incremented by the sum of the current value of i and the product of the current value of j and m. After both loops have fully executed, the final value of a is printed out. Essentially, this code performs a computation based on the two loop counters and accumulates the result in the variable a.

4.1.16 Ex. (37) – Nested for loops & summation based on counters and upper limits (II)

```
a = 0
m = 4

for j in range(1, m+1):
    for i in range(1, j+1):
        a = a + (i + j * m)

print(a)
```

Output:

```
140
```

The given Python code begins by initializing two variables: *a* is set to 0, and *m* is set to 4. Following this, there is a nested loop structure. The outer loop runs with the variable *j*, starting from 1 up to and including the value of *m*, which is 4. Inside this outer loop, there is an inner loop that runs with the variable *i*, starting from 1 and going up to the current value of *j* from the outer loop. Within the inner loop, the code calculates a value by adding *i* and the product of *j* and *m*. This calculated value is then added to the current value of *a*, effectively updating *a* with each iteration of the inner loop. Once both loops have completed their iterations, the final accumulated value of *a* is printed out using the *print*() function.

4.1.17 Ex. (38) – Nested for loops & summation based on counters and upper limits (III)

```
a = 0
m = 5
n = 7

for j in range(1, m + 1):
    for i in range(j, n + 1):
        a += (i + j * m)

print(a)
```

Output:

```
445
```

The given Python code initializes three variables *a*, *m*, and *n* with the values 0, 5, and 7, respectively. The code then has a nested loop where the outer loop runs with the variable *j* iterating from 1 through the value of *m* (which is 5). For each iteration of the outer loop, the inner loop runs with the variable *i* starting from the current value of *j* up to the value of *n* (which is 7). Within the inner loop, the value of *a* is updated by adding the sum of *i* and the product of *j* and *m*. After both loops are completed, the value of *a* is printed. Essentially, this code is calculating a summation based on the provided formula and values of *m* and *n*.

4.1.18 Ex. (39) – Show *i* and *j* coordinates at each step

```
for i in range(2):
    for j in range(3):
        print(f"i = {i}, j = {j}")
```

Output:

```
i = 0, j = 0
i = 0, j = 1
i = 0, j = 2
i = 1, j = 0
i = 1, j = 1
i = 1, j = 2
```

The provided Python code consists of a nested loop. The outer loop, controlled by the variable *i*, runs for two iterations, with *i* taking values 0 and 1 (two is exclusive). Inside each iteration of this outer loop, there is an inner loop, controlled by the variable *j*, which runs for three iterations, making *j* take on the values 0, 1, and 2. During each iteration of the inner loop, the *print* function is called to display the current values of both *i* and *j*. As a result, the message "i = [value of i], j = [value of j]" will be printed a total of six times, reflecting every combination of *i* and *j* within the specified ranges. For example, the first few messages will be "i = 0, j = 0", "i = 0, j = 1", "i = 0, j = 2", and so on.

4.1.19 Ex. (40) – One *for loop* that simulates two for loops

```
i = j = 0
n1 = 2
n2 = 3
q = n1 * n2

for v in range(q):
    j = v % n2
    if j==0 and v!=0 and i<n1 and v!=q:
        i += 1
    print(f"i = {i}, j = {j}")
```

Output:

```
i = 0, j = 0
i = 0, j = 1
i = 0, j = 2
i = 1, j = 0
i = 1, j = 1
i = 1, j = 2
```

The Python code snippet initializes three variables *i* and *j* to 0 and two other variables *n1* and *n2* to 2 and 3, respectively. It also calculates the product of *n1* and *n2*, storing the result in a variable named *q*. A *for-loop* then iterates *v* from 0 up to, but not including, the value of *q*. Inside the loop, the value of *j* is calculated as the remainder of the division of *v* by *n2*. A conditional if statement checks if *j* is 0, *v* is not 0, *i* is less than *n1*, and *v* is not equal to *q*. If all these conditions are met, *i* is incremented by 1. After each iteration of the loop, the *print* function is called to output the current values of *i* and *j*. The purpose of the code is to explore the behavior of the variables *i* and *j* as *v* goes from 0 to *q*, under certain conditions specified in the *if* statement. It showcases how the value of *i* can be incremented based on the other variables and conditions within the loop.

Dynamically Resizable Arrays (Lists) 5

Data structures are pivotal constructs that enable the systematic organization and management of data [1]. One of the quintessential and most universally utilized data structures is the array. A *dynamically resizable array* or a *list*, can be aptly described as a collection of items. Its salient feature is the direct access it offers to any indexed element, granting it significant computational advantages in specific scenarios. Characteristics of **classical arrays** in C++ or Java and so on, encompass homogeneity, implying that all elements within an array are of the same data type, ensuring uniform memory footprint. In these classical computer languages, arrays are static in nature, unlike dynamic data structures found in Python and other modern languages. Each element in the array is associated with a unique index, facilitating swift access to any element with known index. Arrays find applications in various domains within computer science. They are fundamental in sorting algorithms like *QuickSort* and *MergeSort*, often employed for their direct access capabilities. Arrays also serve as foundational elements for complex data structures such as *heaps*, *hash tables*, and *dynamically resizable arrays.* Advantages of arrays include their speed in retrieval operations and efficient memory allocation due to contiguous storage. However, they have limitations, notably their fixed size and the relatively costly nature of insertion and deletion operations, particularly for elements in the middle of the array. However, from here on **we will refer to the *list* as an *array***, since their behavior is basically the same. In other words, lists can hold different types of data and their length can be changed, while classic arrays have a fixed length and contain only one type of data, but, the rest is the same, from indexing to syntax. Note: In my view the name *list* is improper as it leads the mind to a 1-dimensional sequence. Also, to call these data structures *lists* instead of *arrays* is bordering lack of respect for the previous generations of

P. A. Gagniuc, *Coding Examples from Simple to Complex*, Synthesis Lectures on Computer Science, https://doi.org/10.1007/978-3-031-53812-4_5

scientists and engineers. In the following examples, we will explore the common methods and techniques used around *dynamically resizable arrays*, also called *lists* in Python, providing information on how to access and manipulate the data stored in these structures.

5.1.1 Ex. (41) – Array addition

```
# a[i] vector
# a[i][j] = matrix
# a[i][j][x] = tensor
# a[i][j][x][y]...

a = [2, 5, 7]
b = [6, 8]

print(a + b)
```

Output:

```
2,5,76,8
```

The provided code starts with a series of comments that serve to explain the structure of nested arrays, often referred to as vectors, matrices, and tensors. These comments lay out a conceptual hierarchy, illustrating how data can be organized in Python arrays. The a variable is declared as an array with three elements: [2, 5, 7]. In the comments, it is described as a vector, implying a one-dimensional array. The comment *a[i][j] = matrix* suggests the potential for two levels of nesting within this array, implying a two-dimensional structure resembling a matrix. The comment *a[i][j][x] = tensor* extends this hierarchy further, indicating that within this matrix-like structure, there is yet another level of nesting, giving it a three-dimensional structure resembling a tensor. The comment *a[i][j][x][y][...]* suggests the possibility of even deeper levels of nesting, forming higher-dimensional structures. Following these comments, two arrays, a and b, are declared with numeric elements. Next, the $print(a + b)$ line allows to concatenate the two arrays a and b.

5.1.2 Ex. (42) – Extracting individual values from the elements of an array

```
a = [2, 5, 7]
b = [6, 8]
c = a[1] + b[0]
print(c)
```

Output:

```
11
```

The provided code snippet demonstrates a series of operations involving arrays and variable assignments. Three variables are declared: a, b, and c. The a variable is assigned an array [2, 5, 7], while the b variable is assigned another array [6, 8]. These arrays can store multiple values. The key operation occurs when the c variable is assigned a value. It calculates this value by adding together two specific elements from the arrays a and b. In short, it takes the second element from a (which is 5) and the first element from b (which is 6) and adds them together. The result of this addition, 11, is stored in the variable c.

Next, the value of the c variable is shown in the output. Thus, this Python code snippet involves array operations and variable assignments, culminating in the addition of specific elements from two arrays.

5.1.3 Ex. (43) – Adding elements

```
A = []

A.append("a")
A.append("b")
A.append("c")

print(A[0] + A[1] + A[2])
```

Output:

```
abc
```

The example initializes a variable A as an empty array. It then proceeds to assign values to its elements. In this case, the values assigned are strings: “a” to $A[0]$, “b” to $A[1]$ and “c” to $A[2]$. Next, it uses the *print* function to output the concatenation of these three elements. The “ + ” operator here is used for string concatenation, which means it combines the three strings “a,” “b,” and “c” into a single string. Thus, the output of this code will be “abc” when it is executed.

5.1.4 Ex. (44) – Using array literals of different data type

```
A = []
B = []

A = ["a", "b", "c"]
B = [1, 2, 3]

print(A[0] + A[1] + A[2])
print(B[0] + B[1] + B[2])
```

Output:

```
abc
6
```

In this code snippet, we are working with two arrays, A and B. Initially, we declare these arrays as empty. However, these empty declarations are later overwritten with new values. The first array, A, is populated with three string elements: “a”, “b”, and “c”. Each of these elements is enclosed in double quotes and separated by commas. The second array, B, is filled with three numeric elements: 1, 2, and 3. These numeric values are not enclosed in quotes because they are treated as integers. After initializing these arrays, the code proceeds to print out the concatenation of elements within each array. For array A, the concatenation of its elements “a”, “b”, and “c” results in the string “abc”. This string is printed to the console using the *print* function. Similarly, for array B, the concatenation of its elements 1, 2, and 3 results in the numeric value 6 ($1 + 2 + 3$). This numeric value is also printed to the console using the *print* function.

5.1.5 Ex. (45) – Accessing array elements

```
A = ["a", "b", "c"]

x = A[1]
y = A[2]

print(x + y)
```

```
Output:

bc
```

This Python code begins by defining an array (list) called "A," which contains three elements: "a," "b," and "c." The array *A* is initialized with these values. Next, the code declares two variables, *x* and *y*. The variable *x* is assigned the value of the second element in the array *A*, which is "b," since Python arrays are zero-indexed. Similarly, the variable *y* is assigned the value of the third element in the array *A*, which is "c." The code prints the result of concatenating the values of *x* and *y* using the " + " operator. In this case, "*x* + *y*" would result in "bc" because *x* holds "b" and *y* holds "c."

5.1.6 Ex. (46) – Changing values in array elements - swap values or replace

```
A = ["a", "b", "c"]

x = A[1]

A[0] = "d"
A[1] = A[2]
A[2] = x

print(A[0] + A[1] + A[2])
```

```
Output:

dcb
```

The above code begins by defining an array called *A* containing three string elements: "a," "b," and "c." Next, it initializes a variable named *x* and assigns it the value at the index 1 of array *A*, which is "b." Following this, the code proceeds to modify the elements within the array *A*. It assigns the string "d" to the first element at index 0, replaces the second element at index 1 with the value from the third element at index 2, and finally, sets the third element at index 2 to the value of *x*, which is "b." In the last line of code, the *print* function is used to output the concatenation of the elements at indexes 0, 1, and 2 of array *A*. The result of this concatenation would be "dbc" based on the modifications made to the array earlier in the code.

5.1.7 Ex. (47) – Extracting individual values from the elements of an array

```
a = [2, 5, 7]
b = [6, 8]

a[1] -= 1
b[0] -= 1

c = a[1] + b[0]

print(c)
```

```
Output:

9
```

This code begins by declaring two arrays, *a* and *b*, containing the elements [2, 5, 7] and [6, 8], respectively. Afterward, the actual code proceeds by decrementing the second element (index 1) of array *a* and the first element (index 0) of array *b*. This means that after these operations, the *a* array will become [2, 4, 7], and the *b* array will become [5, 8]. Next, a new variable *c* is declared and assigned the value of the sum of the updated second element of array *a* (which is now 4) and the updated first element of array *b* (which is now 5). Therefore, *c* will be assigned the value 9. In summary, this code modifies two arrays, calculates the sum of specific elements from those arrays, and then prints the result to the console.

5.1.8 Ex. (48) – Array length

```
a = [5, 6, 8]
b = len(a)

print(b)
```

```
Output:

3
```

Here, a variable *a* is declared and initialized as an array containing three elements: 5, 6, and 8. This array is created using square brackets [] and the elements are separated by commas. Next, another variable *b* is declared and assigned the value of *len*(*a*). Here, *len*(*a*) is a property of the array *a*, which represents the number of elements in the array. In this case, since there are three elements in the array *a*, *b* will be assigned the value 3. Next, the value of *b* is printed in the output. Thus, this code snippet creates an array *a*, determines its length, and prints that length to the console.

5.1.9 Ex. (49) – Accessing the values from the components of an array

```
A = [1, 2, 3]

if A[0] < A[1]:
    A[2] += 1

print("A[2]=" + str(A[2]))
```

```
Output:

A[2]=4
```

This code snippet begins by declaring a variable named *A* and assigning it an array containing three elements: 1, 2, and 3. Thus, this array is represented as [1, 2, 3]. Next, there is an *if* statement that checks a condition. It evaluates whether the value at the first index of array *A*, which is *A*[0], is less than the value at the second index, *A*[1]. In this case, *A*[0] contains 1, and *A*[1] contains 2, which is indeed true, as 1 is less than 2. When the condition in the *if* statement is true, it increments the value at the third index of array *A*, which is *A*[2]. Thus, *A*[2] + = 1; adds 1 to the existing value in *A*[2], resulting in *A*[2] being updated to 4. Next, it displays a message that includes the updated value of *A*[2]. The message is constructed by concatenating the string "A[2] = " with the value of *A*[2], which is 4. Thus, when this code is executed, it will print "A[2] = 4" to the console.

5.1.10 Ex. (50) – Traverse a 1D array using a *while loop* (I)

```
A = ["a", "b", "c", "d", "e", "f", "g"]

i = 0
t = ''

while i < len(A):
    t += "\n i[" + str(i) + "]=" + A[i]
    i += 1

print(t)
```

Output:

```
i[0]=a
i[1]=b
i[2]=c
i[3]=d
i[4]=e
i[5]=f
i[6]=g
```

This Python code begins by initializing an array called *A*, which contains the elements "a," "b," "c," "d," "e," "f," and "g." Following this, the code sets up two variables, *i* and *t*, both initially empty. Next, *i* is initialized to 0, and *t* is an empty string. The code then enters a *while-loop* that runs as long as the value of *i* is less than the length of the array *A*. Inside the loop, there is an assignment to the variable *t*. The assignment appends a newline character followed by a string constructed from the value of *i* and the corresponding element in the array *A*. This creates a string that looks like "\n i[0] = a", "\n i[1] = b", and so on. After the assignment, *i* is incremented by 1. Once the loop has iterated through all elements in the array *A*, the code prints the value of *t*. In essence, this code is designed to loop through the elements of the array *A*, building the string *t* that contains the output, namely the information about the index *i* and the corresponding element in the array.

5.1.11 Ex. (51) – Traverse a 1D array using a *while loop* (II)

```
A = ["a", "b", "c", "d", "e", "f", "g"]

i = 0
t = ''

while i < len(A):
    t += "\n i[" + str(i) + "]=" + A[len(A)-i-1]
    i += 2

print(t)
```

Output:

```
i[0]=g
i[2]=e
i[4]=c
i[6]=a
```

This Python code initializes an array named A containing seven elements: “a,” “b,” “c,” “d,” “e,” “f,” and “g.” It then declares two variables, i and t, and assigns them initial values of 0 and an empty string, respectively. The code enters a *while-loop*. Inside the loop, there is a string concatenation operation. It appends a newline character followed by a string that includes the current value of i and the complementary element (i.e., $len(A)-i-1$) from the array A. This information is appended to the t string. The i variable is incremented by two with each iteration of the loop, thus always skipping the next element. The loop continues to execute as long as the value of i is less than the length of the array A. Next, the code prints the value of the t variable. Overall, this code builds a string t by concatenating information about each element in the array A along with its index and then prints the resulting string.

5.1.12 Ex. (52) – Traverse a 1D array using a *for loop*

```
A = ["a", "b", "c", "d", "e"]

t = ""

for i in range(len(A)):
    t += "\n A[" + str(i) + "]=" + A[i]

print(t)
```

Output:

```
A[0]=a
A[1]=b
A[2]=c
A[3]=d
A[4]=e
```

This code begins by defining a constant array called A containing five elements, namely “a,” “b,” “c,” “d,” and “e.” Next, there is a declaration of an empty string variable called t. The code then enters a *for-loop* that initializes a loop counter i from 0 up to one less than the length of the array A, which is 5 in this case. Inside the loop, there is a statement that appends a newline character (“\n”) to the t string, followed by the text “A[” concatenated with the current value of i, followed by “] = ” and the value at the corresponding index in the array A. In other words, during each iteration of the loop, it appends a line to the t string that displays the index and value of each element in the A array. Next, after the loop finishes, the implementation prints the t string using the *print* function. This code

processes the elements in the *A* array, creates a formatted string with their indices and values, and then prints the string.

5.1.13 Ex. (53) – The *for a in b*

```
a = ["x", "y", 2]

for b in range(len(a)):
    print(a[b])
```

```
Output:

x
y
2
```

The above code snippet demonstrates the creation of a variable *a* and the utilization of a *for…in loop* to iterate through its elements. Variable *a* is assigned with an array containing three elements: the strings *x* and *y*, and the number 2. Subsequently, a *for…in loop* is used to traverse the elements of this array. Inside the loop, the *print*(*a*[*b*]) statement to displays each element of the array. Thus, the code primary purpose is to showcase the *for…in loop* for iterating over the elements of an array.

5.1.14 Ex. (54) – Print all integers from array using a *for loop*

```
a = [5, 6, 8]

for j in range(len(a)):
    print(a[j])
```

```
Output:

5
6
8
```

This example initializes an array called *a* containing three numeric values: 5, 6, and 8. It then proceeds to iterate through the elements of this array using a *for-loop*. The *for-loop* is configured to start with an index variable *j* set to 0, which corresponds to the first element in the array. It continues looping as long as *j* is less than or equal to the length of the array minus 1 (i.e., *len*(*a*)). This condition ensures that the loop iterates over all the elements in the array. Within the loop, there is a *print* statement. This statement outputs the value at the current index *j* in the array *a*. The loop will run for each element in the array, starting with 5, then 6, and finally 8. Thus, this code snippet is a basic example of how to iterate through an array in Python using a *for-loop* and print each element is value to the console. When executed, it will display the values 5, 6, and 8 in the console window, each on a separate line.

5.1.15 Ex. (55) – Sum all values from array

```
a = [5, 6, 8]
b = 0

for j in range(len(a)):
    b = b + a[j]

print(b)
```

Output:

```
19
```

This code snippet begins by defining two variables. The first variable, *a*, is an array containing three numerical values: 5, 6, and 8. The second variable, *b*, is initialized with the value 0. The code then enters a *for-loop*, which is a control structure used for iterating through the elements of an array. In this loop, a variable *j* is initialized to 0, and the loop continues as long as *j* is less than or equal to the length of array *a* minus 1. The loop iterates through each element of the *a* array. Inside the loop, there is an assignment statement that increments the *b* variable. It adds the current element of the *a* array, which is indexed by *j*, to the *b* variable. This effectively accumulates in the *b* variable the sum of all the elements in the *a* array. Next, the code calls function *print* with the argument *b*. The code calculates the sum of the elements in the *a* array and prints the result to the console.

5.1.16 Ex. (56) – Multiplication involving a scalar and a 1D array

```
a = [5, 6, 8]

for j in range(len(a)):
    a[j] = 2 * a[j]

print(a)
```

Output:

```
10,12,16
```

This code snippet initializes a variable *a*, and then performs a loop operation on it. The variable *a* is an array containing the elements 5, 6, and 8. Next, a *for-loop* iterates through the elements of array *a*. The loop starts with an index variable *j* set to 0 and continues as long as *j* is less than or equal to the length of array *a* minus 1 (i.e., *len*(*a*)). In each iteration of the loop, the value at the *j*-th index of array *a* is doubled (multiplied by 2) and then assigned back to the same position in the array. Once the loop completes, the value of variable array *a* is shown in the console window.

5.1.17 Ex. (57) – Insert values into an array

```
a = []

# In Python, the range end
# is exclusive, so we use
# 11 to include 10.

for j in range(11):
    a.append(j)

print(a)
```

```
Output:

0,1,2,3,4,5,6,7,8,9,10
```

The above code initializes an empty array a using the variable declaration "$a = []$". This array will be used to store a series of values. Next, there is a *for-loop* that iterates from 0 to 10, inclusive, with the variable j starting at 0 and incrementing by 1 in each iteration. Within the loop, the code assigns the value of j to the corresponding index in the array a. This means that during each iteration of the loop, the value of j is added to the a array in a new element. Next, there the contents of the array a is printed to the output. Note that in Python, the range end is exclusive, so we use 11 to include 10.

5.1.18 Ex. (58) – Insert ascending and descending integer values into arrays

```
a = []
b = []

# range(11) to include 10,
# since range in Python is
# upper-bound exclusive.

for j in range(11):
    a.append(j)
    b.append(10 - j)

print("a =", a)
print("b =", b)
```

```
Output:

a = 0,1,2,3,4,5,6,7,8,9,10
b = 10,9,8,7,6,5,4,3,2,1,0
```

This code initializes two arrays, *a* and *b*, as empty arrays. It then enters a *for-loop* that iterates from 0 to 10, inclusive. During each iteration of the loop, it assigns values to the *a* and *b* arrays based on the loop index *j*. Specifically, within the loop, the code sets *a*[*j*] (*a.append* adds one element at each iteration, that is why the index of the new element coincides with *j*) to the current value of *j*, which corresponds to the numbers from 0 to 10. Simultaneously, it sets *b*[*j*] to the value of 10 - *j*, effectively counting down from 10 to 0. These assignments continue for each iteration of the loop. After the loop completes its execution, it prints the contents of both arrays *a* and *b*.

5.1.19 Ex. (59) – Add forward and reverse values and subtract max

```
a = []
b = []
c = []

# In Python, range(11)
# generates numbers from
# 0 to 10 inclusive.

for j in range(11):
    a.append(j)
    b.append(10 - j)
    c.append(a[j] + b[j] - 10)

print("a =", a)
print("b =", b)
print("c =", c)
```

```
Output:

a = 0,1,2,3,4,5,6,7,8,9,10
b = 10,9,8,7,6,5,4,3,2,1,0
c = 0,0,0,0,0,0,0,0,0,0,0
```

This code initializes three arrays, *a*, *b*, and *c*, and then populates them using a *for-loop*. In the beginning, three empty arrays, *a*, *b*, and *c*, are declared to store integer values. The *for-loop* runs from *j* equal to 0 to 10. During each iteration of the loop, the value of *j* is used as an index to populate the arrays *a* and *b*. Specifically, *a*[*j*] is assigned the current value of *j*, and *b*[*j*] is assigned 10-*j*. Subsequently, the array *c* is populated based on the values of *a* and *b*. For each index *j*, *c*[*j*] is calculated as the sum of *a*[*j*], *b*[*j*], and −10. Next, the code prints the values of arrays *a*, *b*, and *c*, displaying their contents as strings along with their respective names.

5.1.20 Ex. (60) – Pointless equilibrium

```
a = []
l = 10

for j in range(l + 1):
    a.append(j + (l - j))

print(a)


# or a second version:


l = 10

# Initialize a list
# of length l+1.

a = [None] * (l + 1)

for j in range(l + 1):
    a[j] = j + (l - j)

print(a)
```

Output:

```
a = 10,10,10,10,10,10,10,10,10,10,10
```

In this code snippet, a program initializes an empty array called a and a variable l with the value 10. The purpose of this code is to populate the array a with values based on the iteration variable j using a *for-loop*. The *for-loop* runs from j equals 0 to l inclusive (that is, $l + 1$). During each iteration of the loop, an expression is evaluated and assigned to the j index of the array a. The expression being assigned consists of two parts: (1) Variable j: This represents the current value of the iteration variable j, and (2) is $(l-j)$ that represents the result of subtracting j from l. These two parts are added together, and the result is stored in the array a at the index j. Essentially, it calculates the sum of j and $(l-j)$ for each j from 0 to l, storing these values in consecutive elements of the array. Thus, after the loop completes execution, the array a will contain values where each element at index j holds the sum of j and $(l-j)$. The resulting array will be $a = [10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10]$. In this case, all elements of the array a are assigned the value 10 because each iteration calculates $j + (l-j)$, and since l is fixed at 10, the sum remains constant across all iterations. Please note also a second version that uses the counter variable j directly to navigate the precalculated list dimensions (i.e., [None] * $(l + 1)$).

5.1.21 Ex. (61) – Max value from array

```
a = [2, 3, 4, 5, 9, 8, 3]
l = len(a)

max_val = 0

for k in range(l):
    if a[k] > max_val:
        max_val = a[k]

print(max_val)
```

Output:

```
9
```

This example initializes an array *a* with a series of numerical values [2, 3, 4, 5, 9, 8, 3]. It then calculates the length of this array and stores it in a variable *l*. Additionally, it initializes a variable *max_val* with an initial value of 0. Next, there is a *for-loop* that iterates over the elements of the array *a*. The loop is controlled by a variable *k*, which starts at 0 and continues until it reaches the value of *l*, which is the length of the array. Within each iteration of the loop, there is an if statement that checks if the current element *a*[*k*] is greater than the current maximum value *max_val*. If *a*[*k*] is indeed greater, it updates the *max_val* variable with the value of *a*[*k*], effectively keeping track of the maximum value encountered in the array. Once the loop has completed, it prints out the maximum value *max_val* to the console window. In essence, this code finds and prints the maximum value present in the array *a*.

5.1.22 Ex. (62) – Min value from array

```
a = [3, 3, 4, 2, 9, 8, 3]
l = len(a)

min_val = a[0]

for k in range(0, l):
    if a[k] < min_val:
        min_val = a[k]

print(min_val)
```

Output:

```
2
```

In the above code, there is an array *a* initialized with values, namely [3, 3, 4, 2, 9, 8, 3]. The variable *l* is assigned the value of the length of the array *a*, which helps determine the range for looping through the array elements. Next, there is the declaration of the variable *min_val*, which is initially assigned the value of the first element of the array *a*, in this case, 3. This variable will be used to store the minimum value found in the array. The code then enters a *for-loop* with the variable *k* starting from 0 and iterating until *k* is one less than *l*, which means it will go through each element in the array. Inside the loop, there is an *if* statement that checks if the current element *a*[*k*] is less than the

current minimum value *min_val*. If *a*[*k*] is indeed smaller than *min_val*, the value of *min_val* is updated to *a*[*k*], effectively finding the minimum value in the array. Next, outside the loop, the code prints the minimum value found in the array using the *print* statement. This code in complementary to the previous one and calculates and prints the minimum value from the array *a* by iterating through its elements and updating the *min_val* variable whenever a smaller value is encountered.

5.1.23 Ex. (63) – Max value above two arrays of the same size

```
a = [2, 3, 4, 5, 9, 8, 3]
b = [1, 2, 3, 4, 5, 6, 7]
l = len(a)

max_value = 0
max_a = 0
max_b = 0

for k in range(l):
    if a[k] > max_a:
        max_a = a[k]
    if b[k] > max_b:
        max_b = b[k]
    if max_a > max_value:
        max_value = max_a
    if max_b > max_value:
        max_value = max_b

print(max_value)
```

```
Output:

9
```

The above code starts by defining two arrays, *a* and *b,* each containing a sequence of numbers. Then, it calculates the length of the array *a* and stores it in variable *l*. Next, it initializes three variables: *max_value*, *maxA*, and *maxB*, all initially set to 0. These variables will be used to keep track of the maximum values in the arrays. The code enters a *for-loop*, where it iterates through the arrays *a* and *b* simultaneously using the loop variable *k*. It begins at index 0 and goes up to one less than *l*. Within the loop, it checks if the value at index *k* in array *a* (*a*[*k*]) is greater than the current maximum value in *maxA*, and if so, updates *maxA* to this new maximum value. Similarly, it checks if the value at index *k* in array *b* (*b*[*k*]) is greater than the current maximum value in *maxB* and updates *maxB* accordingly. After updating *maxA* and *maxB*, the code also checks if either *maxA* or *maxB* is greater than the current maximum value in *max_value*. If either of them is greater, it updates *max_value* to the larger of the two. The code prints the value stored in the *max_value* variable, which represents the maximum value among both arrays *a* and *b*. Thus, this code finds and prints the maximum value from the combined elements of arrays *a* and *b*.

5.1.24 Ex. (64) – Max value above two arrays of different sizes

```
a = [2, 3, 4, 5, 9, 8, 3]
b = [14, 2, 3, 41, 5, 6, 77]
l = [0, 0]

l[0] = len(a)
l[1] = len(b)

r = l[0]
if l[0] < l[1]:
    r = l[1]

max_value = 0

for k in range(r):
    if k < l[0] and max_value < a[k]:
        max_value = a[k]
    if k < l[1] and max_value < b[k]:
        max_value = b[k]

print(max_value)
```

```
Output:

77
```

This code begins by defining two arrays, *a* and *b*, each containing a series of numeric values. An empty array *l* is also initialized for further use. Next, the code assigns the length of arrays *a* and *b* to the elements of the array *l*. Specifically, *l*[0] stores the length of array *a*, and *l*[1] stores the length of array *b*. The code then compares the lengths of arrays *a* and *b* to determine the maximum length and stores it in the variable *r*. If the length of *b* is greater than the length of *a*, variable *r* is assigned the value of the length of *b*; otherwise, it retains the value of the length of *a*. Subsequently, the variable *max_value* is initialized to 0. The code enters a *for-loop* that iterates from 0 to *r* (exclusive). Inside the loop, it checks if the current index *k* is within the bounds of the lengths of arrays *a* and *b*. If so, it compares the value at index *k* in the respective array with the current maximum value stored in *max_value*. If the value at index *k* is greater than the current maximum, *max_value* is updated to hold that value. Next, the code prints the maximum value (*max_value*) to the output. In essence, this code finds and prints the maximum value among the elements of arrays *a* and *b*, considering both arrays up to the length of the longer one.

5.1.25 Ex. (65) – Which is bigger between n and $n + 1$?

```
a = [2, 3, 4, 5, 9, 8, 3, 8, 3]
l = len(a) - 1

t = ''

for k in range(l):
    if a[k] > a[k + 1]:
        t += '>'
    else:
        t += '<'

print(t)
```

Output:

```
<<<<>><><
```

The given source code begins by initializing an array a with a set of numeric values: [2, 3, 4, 5, 9, 8, 3, 8, 3]. Next, it calculates the length of the array minus one and stores it in a variable l. In this case, l becomes 8 since there are nine elements in the array. The code then declares an empty string t, which will be used to accumulate symbols as the code iterates through the array. A *for-loop* is set up to iterate through the elements of the array a. The loop variable k starts at 0 and continues until it reaches the value of l (exclusive). During each iteration, the code checks if the element at index k in the array a is greater than the element at the next index, $k + 1$. If this condition is true, it appends the " > " symbol to the string t, indicating that the current element is greater than the next one. If the condition is false, it appends the " < " symbol to t, indicating that the current element is less than or equal to the next one. After the loop has processed all elements in the array, the code prints the resulting string t to the output. The output will be a sequence of " > " and " < " symbols, indicating the comparison results between consecutive elements in the array. The exact output will depend on the values in the array a.

5.1.26 Ex. (66) – Which is bigger between n and $n + 1$? (optimisation)

```
a = [2, 3, 4, 5, 9, 8, 3, 8, 3]
l = len(a) - 1

t = ' '
r = '<'

for k in range(0, l):
    if a[k] > a[k + 1]:
        r = ">"
    t += r
    r = "<"

print(t)
```

Output:

```
<<<<>><><
```

This code begins by defining an array a containing a sequence of integers, namely [2, 3, 4, 5, 9, 8, 3, 8, 3]. The variable l is then initialized to store the length of the array minus 1, which is the last valid index within the array. Next, two variables t and r are declared and initialized. Variable t is initialized with a space character, and r is initialized with the less-than symbol (" < "). Following the variable declarations, there is a *for-loop* that iterates from k equals 0 up to and excluding l. During each iteration of the loop, it checks if the element at index k in the array a is greater than the element at the next index ($k + 1$). If this condition is true, it assigns the greater-than symbol (" > ") to the variable r. Otherwise, it assigns the less-than symbol (" < ") to r. Once the value of r is determined, it is then appended to the string stored in the variable t. This process continues through each iteration of the loop, building a string in t where each character corresponds to whether the element at the current index is greater than or less than the next element. The code prints the resulting string t to the output, which represents a sequence of " > " and " < " symbols based on the comparisons between adjacent elements in the array a.

5.1.27 Ex. (67) – Sum two arrays

```
a = [2, 3, 4, 5, 9, 8, 3]
b = [1, 2, 3, 4, 5, 6, 7]
c = []
l = len(a) - 1

for k in range(l + 1):
    c.append(a[k] + b[k])

print("c =", c)
```

Output:

```
c = 3,5,7,9,14,14,10
```

This code begins by defining three arrays: a, b, and c. The a array contains the values [2, 3, 4, 5, 9, 8, 3], while the b array holds [1, 2, 3, 4, 5, 6, 7]. An empty array c is also initialized, which will be used to store the result of adding corresponding elements from a and b. The variable l is assigned the value of $len(a)-1$, which represents the length of the a array minus one, effectively representing the last index of the arrays. The code then enters a *for-loop* that iterates through indices from 0 to l. Above each iteration, it adds the elements at the current index k from arrays a and b, and stores the result in the corresponding index k of the c array. This process effectively performs element-wise addition between arrays a and b. At the end of the *for-loop*, the code prints the result by concatenating the string "c = " with the c array, which will display the content of array c after all the additions have been performed.

5.1.28 Ex. (68) – Simple array mapping

```
a = [10, 9, 8, 7, 6, 5, 4, 3, 2, 1, 0]
b = [1, 1, 1, 2, 2, 2, 1, 1, 1, 1, 1]
c = []

for j in range(11):
    c.append(a[b[j]])

print("c =", c)
```

Output:

```
c = 9,9,9,8,8,8,9,9,9,9,9
```

The given code snippet begins by defining the same three arrays: a, b, and an empty array c. Array a contains 11 integer elements in descending order from 10 to 0. Array b contains 11 integer elements, which represent indices or positions in array a. The values in array b suggest a pattern where some indices may repeat. An empty array c is declared to store the results. The code then enters a *for-loop* that iterates from 0 to 10, using the variable j as the loop counter. Inside the loop, a new element is added to c, namely element j. Thus, the value of $c[\,j]$ is assigned a value from array a at the index specified by $b[\,j]$. This means that for each iteration of the loop, $c[\,j]$ is assigned the value from a at the position specified by $b[\,j]$. The code then prints the contents of array c to the console, displaying the result of this operation. Thus, this code performs a series of value assignments from array a to array c based on the indices specified in array b, and then it prints the resulting array c to the console.

5.1.29 Ex. (69) – Sum by coordinates (I)

```
a = [2, 3, 4, 5, 9, 8, 3]
b = [1, 2, 3, 4, 5, 6, 7]
c = [1, 1, 1, 4, 4, 4, 6]
l = len(a) - 1

for k in range(0, l + 1):
    c[k] = a[c[k]] + b[k]

print("c =", c)
```

Output:

```
c = 4,5,6,13,14,15,10
```

In this code, three arrays a, b, and c are defined with initial values. Array a contains the values [2, 3, 4, 5, 9, 8, 3], array b contains [1, 2, 3, 4, 5, 6, 7], and array c is initially set to [1, 1, 1, 4, 4, 4, 6]. The variable l is defined to store the length of array a minus one. The code then enters a *for-loop* with the variable k ranging from 0 to l, inclusive (i.e., $l-1$). Inside the loop, each element of array c at index k is updated. Namely, $c[k]$ is assigned the value of $a[c[k]] + b[k]$. This means that for each element $c[k]$, it looks up the value at the same index in array a, adds the corresponding value from array b, and stores the result back in array c at index k. This process is repeated for all elements in the specified range of k. At the end of the cycle, the code prints the updated array c using

print("c = " + *c*);. Thus, this code modifies the values in array *c* based on the values in arrays *a* and *b* by using their indices and then displays the updated *c* array.

5.1.30 Ex. (70) – Sum by coordinates (II)

```
a = [2, 3, 4, 5, 9, 8, 3]
b = [1, 2, 3, 4, 5, 6, 7]
c = [1, 1, 1, 4, 4, 4, 6]
l = len(a) - 1

for k in range(l + 1):
    c[k] = a[c[k]] + b[c[k]]

print("c =", c)
```

Output:

```
c = 5,5,5,14,14,14,10
```

In this example, there are three arrays: *a*, *b*, and *c*, each containing a series of integer values. The arrays *a* and *b* are initialized with values, and *c* initially holds a set of integer values. The variable *l* is assigned the value of $len(a)-1$, which represents the index of the last element in the array *a*. The code enters a *for-loop* that iterates from *k* equal to 0 to *l*, inclusive (i.e., $l-1$). Inside the loop, the elements of the *c* array are modified based on the values in arrays *a* and *b*. For each *k* in the loop, the value at index *k* in array *c* is updated to be the sum of $a[c[k]]$ and $b[c[k]]$. Next, outside the loop, the result is printed to the console, showing the updated values in array *c*. This code essentially modifies the *c* array by adding values from arrays *a* and *b* based on the indices specified in *c*, and it prints the resulting array *c* to the console.

5.1.31 Ex. (71) – Cutoff value

```
a = [2, 3, 4, 5, 9, 8, 3]
b = [1, 2, 3, 4, 5, 6, 7]
c = [1, 1, 1, 4, 4, 4, 6]
l = len(a) - 1

for k in range(0, l + 1):
    if a[c[k]] + b[c[k]] > 5:
        c[k] = a[c[k]] + b[c[k]]
    else:
        c[k] = 0

print("c =", c)
```

Output:

```
c = 0,0,0,14,14,14,10
```

In the above code, there are three arrays *a*, *b*, and *c*, each containing a sequence of numeric values. The length of array *a* is stored in the variable *l*. The code then enters a loop that iterates from 0 to *l*. Inside the loop, for each value of *k*, it calculates the sum of $a[c[k]]$ and $b[c[k]]$. If this sum is greater than 5, it assigns this sum to $c[k]$. Otherwise,

it sets $c[k]$ to 0. Next, after the loop completes, it prints the resulting array c, which has been modified based on the conditions described above.

5.1.32 Ex. (72) – Swap array elements by pattern

```
# Swap array elements by pattern

a = [2, 3, 4, 5, 9, 8, 3]
b = [1, 2, 3, 4, 5, 6, 7]
c = [0, 1, 1, 0, 0, 0, 1]
l = len(a)

for k in range(l):
    t = 0
    if c[k] == 1:
        t = a[k]
        a[k] = b[k]
        b[k] = t

print("a =", a)
print("b =", b)
```

Output:

```
a = 2,2,3,5,9,8,7
b = 1,3,4,4,5,6,3
```

This code swaps elements between two arrays a and b based on a corresponding pattern defined in array c. The code begins by defining three arrays: a, b, and c, each containing a set of values. These arrays represent the data that will be manipulated. The variable l is assigned the length of array a, which is used as the loop termination condition in the subsequent *for-loop*. Inside the *for-loop*, a counter variable k is used to iterate through each element of the arrays. Within the loop, a temporary variable t is initialized to 0. This variable will be used for temporarily storing values during the swap operation. An if statement checks if the value of $c[k]$ is equal to 1. If $c[k]$ is indeed equal to 1, it means that a swap operation should be performed for the current elements. Inside the if block, the values of $a[k]$ and $b[k]$ are swapped using the temporary variable t. This is done to exchange the corresponding elements of arrays a and b when the pattern in array c dictates it. After the *for-loop* completes execution, the code prints the updated arrays a and b to the console, showing the result of the swapping operation. Note that this code swaps elements between arrays a and b based on the pattern defined in array c, effectively modifying the contents of arrays a and b accordingly.

5.1.33 Ex. (73) – Mix array based on pattern

```
a = [2, 3, 4, 5, 9, 8, 3]
b = [1, 2, 3, 4, 5, 6, 7]
c = [0, 1, 1, 0, 0, 0, 1]
l = len(a)

for k in range(l):
    if c[k] == 1:
        c[k] = a[k]
    else:
        c[k] = b[k]

print("c =", c)
```

Output:

```
c = 1,3,4,4,5,6,3
```

The example combines two arrays, a and b, into a new array c based on a pattern defined by the c array. The a array contains the elements [2, 3, 4, 5, 9, 8, 3], and the b array contains [1, 2, 3, 4, 5, 6, 7]. Additionally, there is a c array with binary values [0, 1, 1, 0, 0, 0, 1]. The goal is to create a new array c with the same length as a and b and populate it based on the pattern found over c. A *for-loop* iterates through each index from 0 to the length of the arrays (l), and at each iteration it checks the corresponding element in the c array ($c[k]$). If $c[k]$ is equal to 1, it assigns the value from the array a at index k, to $c[k]$. Also, if $c[k]$ is not equal to 1 (i.e., 0), it assigns the value from the b array at index k to $c[k]$. After the loop completes, the resulting c array contains elements that are either from a or b depending on the pattern defined in c. At the end, it prints the contents of the c array, displaying the mixed array based on the pattern.

5.1.34 Ex. (74) – Swap array values

```
a = ["a", "a", "a", "a", "a", "a"]
b = ["b", "b", "b", "b", "b", "b"]
l = len(a) - 1

# Swapping the array values.

for k in range(0, l + 1):
    t = a[k]
    a[k] = b[k]
    b[k] = t

print("a =", a)
print("b =", b)
```

Output:

```
a = b,b,b,b,b,b
b = a,a,a,a,a,a
```

The provided example is designed to swap values between two arrays, a and b, using a loop. Two arrays, a and b, are initially defined with six identical elements each, represented as strings. Additionally, a variable l is declared and assigned the value of the length of array a minus 1. The core of the code lies within a *for-loop* that iterates from 0 to l, inclusive, using the loop variable k. Within this loop the current element of a at index

k is temporarily stored in a variable *t*. The element at index *k* in array *a* is replaced with the corresponding element from array *b*. The element at index *k* in array *b* is assigned the value stored in the temporary variable *t*. This effectively swaps the values of the two arrays at position *k*. After the loop completes, the swapped arrays *a* and *b* are printed to the console window, clearly indicating the changes that occurred as a result of the swap operation.

5.1.35 Ex. (75) – Intermittent value swap

```
# ziperr - intermittent value swap

a = ["a", "a", "a", "a", "a", "a"]
b = ["b", "b", "b", "b", "b", "b"]
l = len(a) - 1

k = 0
while k <= l:
    k += 1
    t = a[k]
    a[k] = b[k]
    b[k] = t
    k += 1

print("a =", a)
print("b =", b)
```

Output:

```
a = a,b,a,b,a,b,a,
b = b,a,b,a,b,a,b,
```

This code demonstrates an intermitent value swap operation between two arrays, *a* and *b*. Initially, both arrays, *a* and *b*, are defined and filled with identical values, where *a* contains multiple instances of *a* and *b* contains multiple instances of *b*. The variable *l* is assigned the value of the length of array *a* minus 1, which determines the limit for the loop. Inside the *for-loop*, the code iterates through the indices of the arrays from 0 to *l* (inclusive) Within each iteration, the loop counter *k* is incremented by 1, effectively skipping every other index. Then, the code performs a swap operation between the elements at the current *k* index in arrays *a* and *b*. This swap operation exchanges the values of *a*[*k*] and *b*[*k*]. Next, the code prints the contents of arrays *a* and *b* after the intermitent value swap operation has been completed, displaying the updated contents of both arrays. Thus, this code swaps values between the *a* and *b* arrays, but it does so intermittently by skipping every other index during the swap operation. As a result, the *a* and *b* arrays will have their values partially exchanged based on this pattern.

5.1.36 Ex. (76) – Reverse string

```
# a = 'abcdef'
# b = list(a)

b = ['a', 'b', 'c', 'd', 'e', 'f']
n = len(b)
c = [None] * n

for i in range(n):
    c[i] = b[n - i - 1]

print(c)
```

Output:

```
f,e,d,c,b,a
```

The following code snippet begins by defining an array b with elements a, b, c, d, e, f, essentially creating an array containing individual characters. Next, it calculates the length of the array b and stores it in the variable n. In this case, n will be equal to 6, as there are six elements in the array b. Then, a new empty array c is declared. This array c will be used to store the elements of array b in reverse order. The code enters a *for-loop* that iterates from $i = 0$ to $i = n-1$ (inclusive). Inside the loop, it assigns the value of $b[n-i-1]$ to the corresponding index in array c. This effectively reverses the order of the elements from array b and stores them in array c. Next, the $print(c)$ statement is used to display the contents of the reversed array c in the output (console window). Thus, this code takes an array b, which contains characters, and reverses the order of its elements, storing the reversed elements in a new array c.

5.1.37 Ex. (77) – The welding of array values

```
# intermitent melting

a = [1, 2, 3, 4, 5, 6, 7]
b = [2, 2, 2, 2, 2, 2, 2]
l = len(a) - 1

k = 0
while k < l:
    k += 1
    a[k] = a[k] + b[k]
    b[k] = a[k]
    k += 1

print("a =", a)
print("b =", b)
```

Output:

```
a = 1,4,3,6,5,8,7
b = 2,4,2,6,2,8,2
```

This code performs a series of operations on two arrays, a and b, both of which initially contain elements from 1 to 7. The code begins by initializing two arrays, a and b, where a contains elements from 1 to 7, and b contains only the number 2 repeated seven times. Next, it calculates the length of the array a ($len(a)-1$) and stores it in variable l. The

code then enters a *while-loop*, where it iterates through the indices of the arrays from 0 to *l*. Inside the loop variable *k* is incremented by 1. This means that *k* will skip every other index in the loop. The element at index *k* of array *a* is updated by adding the element at the same index *k* of array *b* to it. This effectively accumulates the values from array *b* into array *a* for the skipped indices. The element at index *k* of array *b* is then updated to match the new value of *a*[*k*]. This synchronizes the values in arrays *a* and *b* at the skipped indices. Once the loop is completed, the code prints out the contents of both arrays *a* and *b*. Thus, this code performs an "intermittent melting" operation by skipping every other index in the arrays *a* and *b*, updating the values in *a* with the corresponding values in *b*, and then synchronizing the values in *b* with the new values in *a*. The result is printed to the console.

5.1.38 Ex. (78) – Static *modulo* - fill up array with *modulo* results

```
a = [10, 9, 8, 7, 6, 5, 4, 3, 2, 1, 0]
b = []

for j in range(11):
    b.append(a[j] % 3)

print("c =", b)
```

```
Output:

c = 1,0,2,1,0,2,1,0,2,1,0
```

The above code begins by declaring two arrays, *a* and *b*. The array *a* is initialized with ten integer values in descending order from 10 to 0. The array *b* is initialized as an empty array. The code then enters a *for-loop* with the variable *j* starting from 0 and continuing until it reaches 10 (inclusive). Within this loop, each element of the *b* array is assigned a value calculated as the remainder of dividing the corresponding element from the *a* array (*a*[*j*]) by 3. This operation effectively computes the modulo 3 of each element in *a* and stores the result in the corresponding position in *b*. Next, the code prints the value of *b* using the *print* function.

5.1.39 Ex. (79) – Dynamic *modulo* - take *a*[*i*] modulo *j*

```
a = [10, 9, 8, 7, 6, 5, 4, 3, 2, 1, 0]
b = []

for j in range(11):
    b.append(a[j] % (j + 1))

print("c =", b)
```

```
Output:

c = 0,1,2,3,1,5,4,3,2,1,0
```

In this snippet, we have two arrays *a* and *b* initialized, and a loop is used to perform some operations and populate the *b* array based on the values in *a*. Two arrays, *a* and *b*, are declared. Variable *a* contains eleven integer values ranging from 10 to 0, and *b* is initially

an empty array. A *for-loop* is set up with the variable j ranging from 0 to 10 (inclusive). This loop will iterate a total of 11 times. Inside the loop, there is an assignment statement. For each iteration, the value at index j in array a is taken ($a[\,j]$), and the modulo operator (%) is applied to it. The divisor in the modulo operation is $(\,j\,+\,1)$. The result of this operation is then assigned to the corresponding index j in array b. Essentially, the code calculates the remainder when the value in a is divided by $j + 1$, and stores that remainder in b. Next, outside the loop, a *print* statement is used to display the contents of array b. It creates a string concatenating "c = " and the array b. This will display the values of array b as a string with "c = " as a prefix.

5.1.40 Ex. (80) – Convert a string to an array

```
a = "0|13|55|56|1|30|123"
b = "5|33|55|90|1|22|127"

aa = a.split("|")
bb = b.split("|")
cc = []

for i in range(len(aa)):
    cc.append(int(aa[i]) + int(bb[i]))

print(cc)
```

```
Output:

5,46,110,146,2,52,250
```

The code from above performs a series of operations on two given strings a and b. These strings contain numerical values separated by the "|" character. The code aims to split these strings into arrays, calculate the sum of corresponding elements from both arrays, and store the results in a new array *cc*, which are then printed in the output. The code initially begins by defining two strings, a and b, each containing a series of numerical values separated by "|" characters. Next, it splits the strings a and b into arrays using the split("|") method. This operation separates the numerical values at each "|" character and stores them in the arrays *aa* and *bb*, respectively. Afterward, an empty array *cc* is declared to store the results of the element-wise addition of the values from aa and *bb*. A

for-loop is used to iterate through the elements of *aa*. The loop runs from $i = 0$ to $i <= len(aa)-1$, ensuring that it goes through all elements in the arrays. During each iteration, the code converts the elements at the current index i in both *aa* and *bb* to numbers using *int*() and then adds them together. The result is stored in the *cc* array at the same index i. Essentially, this loop calculates the element-wise sum of the corresponding elements from *aa* and *bb* and populates the *cc* array with the results. Next, the code prints the contents of the *cc* array, which now holds the summed values of the corresponding elements from the original *a* and *b* strings.

5.1.41 Ex. (81) – The rule of three simples

```
a = [5, 1, 8, 4, 6, 2, 9, 8]

n = len(a)
max_value = 0
m = 100
t = []

for i in range(n):
    if a[i] > max_value:
        max_value = a[i]

for i in range(n):
    p = (m / max_value) * a[i]
    print(f'{p}%')
```

Output:

```
55.55555555555556%
11.11111111111111%
88.88888888888889%
44.44444444444444%
66.66666666666666%
22.22222222222222%
100%
88.88888888888889%
```

This code begins by defining an array *a* with a series of numeric values. It calculates the length of the array *n* and initializes some variables: *max_value* is set to 0, *m* is assigned the value 100, and an empty array *t* is created. The first *for-loop* iterates through the elements of the array *a*. Within this loop, there is an if statement that checks if the current element $a[i]$ is greater than the current maximum value *max_value*. If it is, the *max_value* variable is updated to the value of $a[i]$, essentially finding the maximum value in the array. After finding the maximum value, a second *for-loop* goes through the elements of the array *a* again. Inside this loop, it calculates a new variable *p* by multiplying *m* by the ratio of the current element $a[i]$ to the maximum value *max_value*. This is done to scale the values in the array *a* proportionally based on the maximum value. Next, the code prints out the calculated *p* value followed by the "%" symbol. This code essentially scales the values in array *a* so that they represent percentages relative to the maximum value found in the array. The scaled values are then printed to the console.

5.1.42 Ex. (82) – Average, standard deviation and coefficient of variation

```
a = [5, 6, 2, 9, 44, 200]
n = len(a)

b = 0
e = 0

for j in range(n):
    b += a[j]

x = b / n

for j in range(n):
    e += (a[j] - x) ** 2

s = (e / (n - 1)) ** 0.5
c = s / x

print('AV =', x)
print('SD =', s)
print('CV =', c)
```

Output:

```
AV = 44.333333333333336
SD = 77.8322983514342
CV = 1.7556157522879894
```

This source code is fundamental for any scientist or/and engineer, and it calculates and prints three statistical measures (average, standard deviation, and coefficient of variation) for a given array of numbers. The code begins by defining an array a containing a set of numerical values. It also initializes two variables, b and e, to zero. Variable b will be used to calculate the sum of all values in the array, and e will be used to calculate the sum of squared differences from the mean. Next, the code calculates the length of the array a and stores it in the variable n. A *for-loop* is used to iterate through the elements of the array a. Inside the loop, each element of the array is added to the variable b, effectively summing up all the values in the array. After the first loop, the code calculates the mean (average) x by dividing the sum b by the length n of the array a. A second *for-loop* is used to iterate through the elements of the array a once again. Inside this loop, the code calculates the sum of squared differences (e) from the mean for each element of the array. Once the second loop is complete, the code proceeds to calculate the standard deviation s using the formula for sample standard deviation. It takes the square root of e divided by $(n-1)$.

Next, the code calculates the coefficient of variation c by dividing the standard deviation s by the mean x. The results are then printed to the console using the *print* function, displaying the average (AV), standard deviation (SD), and coefficient of variation (CV) along with their respective values. Thus, this code performs basic statistical calculations for a given array of numbers, providing insights into the central tendency, dispersion, and relative variability of the data.

5.1.43 Ex. (83) – Horizontal chart from ASCII characters

```
a = [5, 1, 8, 4, 6, 2, 8, 9]

c = '#'
t = ''

n = len(a)

for i in range(n):
    for k in range(a[i]):
        t += c

    t += '\n'

print(t)
```

Output:

```
#####
#
########
####
######
##
########
#########
```

The above example initializes an array a with a sequence of integers. It also defines two strings, c and t, with initial values. The variable c is set to "#", and t is an empty string. The length of array a is stored in variable n. The code then enters a nested loop structure. The outer loop, controlled by variable i, iterates through each element of the array a. Inside this loop, there is an inner loop controlled by the variable k. The inner loop runs a number of times equal to the value at the current index of $a[i]$. During each iteration of the inner loop, the character "#" is appended to the string t. After the inner loop finishes for a specific i, a newline character "\n" is appended to the string t. This creates a pattern of "#" characters on each line, where the number of "#" characters on a line is determined by the value at the current index of $a[i]$. Thus, the t string, containing the pattern of "#" characters, is printed to the console.

5.1.44 Ex. (84) – Horizontal chart with bars proportional with max from array

```
a = [5, 2, 8, 4, 6, 22, 8, 9]

m = 15
c = '#'
t = ''
max_value = 0

n = len(a)

for i in range(n):
    if a[i] > max_value:
        max_value = a[i]

for i in range(n):
    f = int((m / max_value) * a[i])

    for k in range(f):
        t += c
    t += '\n'

print(t)
```

Output:

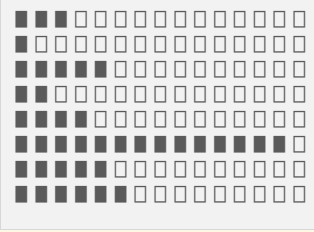
```
###
#
#####
##
####
###############
#####
######
```

The above example performs several operations on an array *a* containing numeric values, and then generates a visual representation of the data using characters. First, an array *a* is defined, which contains the numbers [5, 2, 8, 4, 6, 22, 8, 9]. Next, some variables are initialized: (1) Variable *m* is set to 15. (2) Variable *c* is set to "#", representing a character used to visualize the data. (3) Variable *t* is an empty string that will accumulate characters to create a visual representation. (4) Variable *max_value* is initialized to 0 and will be used to find the maximum value in the array *a*. (5) Variable *n* is assigned the length of the array *a*. The code then enters a loop to find the maximum value in the array *a*. It iterates through each element of *a* using a *for-loop*, and if the current element (*a*[*i*]) is greater than the current maximum (*max_value*), it updates the *max_value* variable with the value of *a*[*i*]. This loop effectively finds the maximum value in the array. The detection of the maximum value, allows the code to enter another loop to generate a visual representation of the data. It iterates through each element of *a* again, using a *for-loop* with *i* as the loop variable. Inside this loop variable *f* is calculated as *int*((*m*/*max_value*) * *a*[*i*]). This computes a scaled value of *a*[*i*] relative to the maximum value (*max_value*) and scales it to fit within the range of 0 to *m*. The *int* function ensures that *f* is an integer. Another nested *for-loop* with *k* as the loop variable is used to append *f* copies of the character *c* to the string *t*. This creates a visual representation where the number of characters represents the scaled value of *a*[*i*]. Once *f* characters are added, a newline character "\n" is injected to start a new line. Next, the code prints the accumulated string *t*, which displays the visual representation of the scaled values of the elements in the array a using the character "#". The number of "#" characters on each line corresponds to the scaled value of the corresponding element in the array *a*.

5.1.45 Ex. (85) – Horizontal chart with UTF characters proportional with max array

```
a = [5, 2, 8, 4, 6, 22, 8, 9]

m = 15
t = ''
max_value = 0

n = len(a)

for i in range(n):
    if a[i] > max_value:
        max_value = a[i]

for i in range(n):

    f = int((m / max_value) * a[i])

    for k in range(m):

        if k < f:
            t += '■'
        else:
            t += '□'

    t += '\n'

print(t)
```

Output:



This code is similar to the previous one, and performs several operations on an array *a* and generates a UTF-8 text-based bar chart, where the length of each bar is proportional to the corresponding element in the array *a* concerning the maximum value in the array. An array *a* is defined with a set of numeric values. Variables *m*, *t*, and *max_value* are initialized. Variable *m* represents the total number of characters for each bar in the chart, *t* will store the final text result, and *max_value* will keep track of the maximum value in the array. The length of the array *a* is determined and stored in variable *n*. A *for-loop* is used to find the maximum value in the array. It iterates through each element of *a*, and if the current element is greater than the current *max_value*, it updates *max_value* with the current element. Another *for-loop* is used to create the bar chart. It iterates through each element of *a* again. Inside the second loop, a variable *f* is calculated. It represents the length of the current bar and is calculated as (*m*/*max_value*) * *a*[*i*], which scales the length of the bar based on the ratio of the current element value to the maximum value in the array. Within a nested loop (*for-loop* with variable *k*), the code checks if *k* is less than *f*. If *k* is less than *f*, it appends a filled square character ("■") to the string *t*, indicating the filled portion of the bar. Otherwise, it appends an empty square character ("□") to represent the empty portion of the bar. After each bar is constructed, the code adds a newline character to *t* to start a new line for the next bar. Next, the resulting *t* variable, containing the text-based bar chart, is printed to the console. This

code essentially visualizes the values in the array *a* by representing them as bars in a simple text format, where the length of each bar is proportional to the value it represents concerning the maximum value in the array.

Traversal of Multidimensional Arrays

6

Traversal of multidimensional arrays is a fundamental operation in computer programming, particularly when dealing with complex data structures and matrices [1]. A multidimensional array is essentially an array of arrays, where each element can be another array. This arrangement allows for the representation of data in a grid or matrix-like format, making it well-suited for various applications, including image processing, data analysis, and simulations. Traversing or iterating through multidimensional arrays involves systematically visiting each element within the structure. This process is crucial for performing tasks such as data manipulation, searching for specific values, or performing mathematical operations across the array. Depending on the programming language, there are various techniques and loops that can be used for efficient multidimensional array traversal. In the following examples, we will explore the common methods and techniques used to traverse multidimensional arrays in Python, providing insights into how to access and manipulate data stored in these complex structures. Whether one is working with two-dimensional arrays, three-dimensional arrays, or even higher-dimensional arrays, understanding how to traverse them is a fundamental skill for a wide range of software development and data analysis tasks. The following examples show different cases. Some classic examples are presented first, using nested loops for multidimensional array traversals, and other more interesting cases show the same types of traversals using one *for-loop* and mathematical formulas for guidance.

P. A. Gagniuc, *Coding Examples from Simple to Complex*, Synthesis Lectures on Computer Science, https://doi.org/10.1007/978-3-031-53812-4_6

6.1.1 Ex. (86) – Accessing the elements of matrix *A*

```
A = [
    ["a", 88, 146],
    ["b", 34, 124],
    ["c", 96, 564],
    [100, 12, "d"],
    ]

print(A[1][2])
```

Output:

```
124
```

The above statement is working with a 2D array named *A*. This array is composed of several sub-arrays, each of which contains a mix of numbers and strings. First, in this code we define a 2D array *A*. This array is structured as an array of arrays. Each sub-array represents a row of data, and within each sub-array, we have elements that can be either strings or numbers. The array *A* contains four sub-arrays, and each of them holds a combination of string and number values: The first sub-array contains three elements: the string "a", the number 88, and the number 146. The second sub-array contains three elements: the string "b", the number 34, and the number 124. The third sub-array contains three elements: the string "c", the number 96, and the number 564. The fourth sub-array contains three elements: the number 100, the number 12, and the string "d". After defining the 2D array *A*, the code prints an element from within the array. Specifically, it prints *A*[1][2]. This notation means that it accesses the second sub-array within *A* (arrays are zero-indexed in Python), and from that sub-array, it retrieves the third element, which is the number 124. Thus, when this code is run, it will output 124 to the console.

6.1.2 Ex. (87) – Accessing the elements of matrix *A* using nested for loops

```
A = [
    ["a", 88, 146],
    ["b", 34, 124],
    ["c", 96, 564],
    [100, 12, "d"],
    ]

t = ""

for i in range(len(A)):
    for j in range(len(A[i])):
        t += "\n A[{}][{}]={}".format(i,j, A[i][j])

print(t)
```

Output:

```
A[0][0]=a
A[0][1]=88
A[0][2]=146
A[1][0]=b
A[1][1]=34
A[1][2]=124
A[2][0]=c
A[2][1]=96
A[2][2]=564
A[3][0]=100
A[3][1]=12
A[3][2]=d
```

This code initializes a two-dimensional array A containing various values, including strings and numbers. The array consists of four subarrays, each containing three elements. These subarrays represent rows and contain a mix of string and numeric values. Next, it declares an empty string variable t which will be used to store the result of the iteration over the A array. The code proceeds to iterate through the array A using nested for loops. The outer loop iterates through the rows of the A array using the variable i, and the inner loop iterates through the elements of each row using the variable j. Within the nested loops, the code appends information about each element of the A array to the string t. It creates a string in the format "$A[i][j] =$ value", where i and j are the indices of the element, and value is the actual value stored in the A array at that position. Thus, the code prints the accumulated string t to the console, displaying the information about each element in the A array, including their positions within the array.

6.1.3 Ex. (88) – Traverse a matrix with a single *for loop* (I)

```
m = [
    [2, 4, 6],
    [3, 5, 6],
    [3, 5, 4]
    ]

i = j = 0
n1 = len(m)
n2 = len(m[0])
q = n1 * n2

for v in range(q):
    j = v % n2
    if j==0 and v!=0 and i<n1 and v!=q:
        i += 1
    print("m[{}][{}]={}".format(i,j,m[i][j]))
```

Output:

```
m[0][0]=2
m[0][1]=4
m[0][2]=6
m[1][0]=3
m[1][1]=5
m[1][2]=6
m[2][0]=3
m[2][1]=5
m[2][2]=4
```

This source code defines a 2D array m, initializes several variables, and then uses a loop to iterate through the elements of the array m. The 2D array m is defined as a 3 $\times$ 3 matrix with specific integer values. Two variables i and j are initialized to 0, and two variables $n1$ and $n2$ are assigned the lengths of the matrix m (number of rows and columns, respectively). The variable q is calculated as the product of $n1$ and $n2$. A *for-loop* is used to traverse the elements of the matrix. The loop runs from $v = 0$ to $v < q$. Within the loop, variable j is calculated as the remainder of v divided by $n2$, which effectively represents the column index within the matrix. An *if* condition checks if j is 0 (indicating a new row) and v is not 0 (to exclude the very first iteration) and if i is less than $n1$ (indicating there are more rows to traverse), and v is not equal to q. If this condition is met, i is incremented by 1, which signifies moving to the next row in the matrix. Next, a *print* statement is used to display the value of the matrix element at the current indices i and j in the format "m["+i+"]["+j+"]=" + $m[i][j]$. In essence, this code

iterates through the elements of the 2D array m row by row, printing out each element along with its row and column indices.

6.1.4 Ex. (89) – Traverse a matrix with a single *for loop* (II)

```
A = [
    ["a", 88, 146],
    ["b", 34, 124],
    ["c", 96, 564],
    [100, 12, "d"],
    ]

t = ""
n = len(A)    # rows
m = len(A[0]) # columns

i = 0
j = 0

for v in range(n * m):
    j = v % m

    if v != 0 and j == 0:
        i += 1

    t += f"{v} A[{i}][{j}]={A[i][j]}\n"

print(t)
```

Output:

```
0 A[0][0]=a
1 A[0][1]=88
2 A[0][2]=146
3 A[1][0]=b
4 A[1][1]=34
5 A[1][2]=124
6 A[2][0]=c
7 A[2][1]=96
8 A[2][2]=564
9 A[3][0]=100
10 A[3][1]=12
11 A[3][2]=d
```

The code defines a 2D array A containing a mix of strings and numbers. It initializes some variables and then enters a loop to iterate through each element of the 2D array and print its position and value. First, the code initializes an empty string t which will be used to accumulate the output. It also determines the number of rows n and columns m in the array A. Then, it sets up two counters, i and j, to keep track of the current row and column in the array. The code enters a nested loop that runs $n \times m$ times, where n is the number of rows and m is the number of columns in the array. Within the loop, j is updated to the current column index by taking the remainder of v (the loop counter) divided by m. An if statement checks if v is not zero (i.e., we are not at the first element of the array) and if j is zero (i.e., it reached the end of a row). If this condition is met, i

is incremented to move to the next row. The code then constructs a string *t* that contains the current element (*v*), the coordinate position of the element on the matrix (*A*[*i*][*j*]) and the value found on it, where *i* and *j* represent the row and column indices, and appends it to the *t* string. Next, a newline character (\n) is added to separate each information of the elements. After the loop finishes, the accumulated string *t* is printed to the console. In essence, this code is used to display the indices and values of all elements in the 2D array *A*, with each element position in the array denoted by *A*[*i*][*j*].

6.1.5 Ex. (90) – Accessing the elements of a 3D array

```
A = [
    [
        ["a", 88, 146],
        ["b", 34, 124],
        ["c", 96, 564],
        [100, 12, "d"],
    ],
    [
        ["e", 48, 996],
        ["f", 34, 554],
        ["g", 26, 884],
        [111, 92, "h"],
    ]
]

print(A[1][2])
```

Output:

```
g,26,884
```

This code defines a 3D array called *A* that contains nested arrays with multiple elements. Each of these nested arrays represents a 2D array, and they are grouped within the larger 3D structure. Inside the 3D array *A*, there are two main levels of nested arrays. The first level contains two sub-arrays, and the second level contains arrays with a mix of strings and numbers as their elements. For example, the first nested array *A*[0] contains four sub-arrays. Each of these sub-arrays consists of three elements. The elements include strings like “a,” “b,” “c,” and “d,” as well as numeric values like 88, 146, 34, 124, 96, 564, 100, and 12. Some sub-arrays even contain a mix of both strings and numbers. Similarly, the second nested array *A*[1] also contains four sub-arrays with similar structures. The last line of code, *print*(*A*[1][2]), is trying to access an element within the *A* array. Specifically, it accesses the third sub-array ([2]) within the second nested array ([1]) and print its contents. In this case, it would print the sub-array ["g", 26, 884].

6.1.6 Ex. (91) – Traverse a 3D object with a single *for loop*

```
A = [
    [
        ["a", 55, 146],
        ["b", 34, 124],
        ["c", 96, 564],
        [100, 12, "d"],
    ],
    [
        ["e", 88, 146],
        ["f", 34, 124],
        ["g", 96, 564],
        [100, 12, "h"],
    ],
    [
        ["i", 88, 146],
        ["j", 34, 124],
        ["k", 96, 564],
        [100, 12, "k"],
    ],
    [
        ["m", 88, 146],
        ["n", 34, 124],
        ["o", 96, 564],
        [100, 12, "p"],
    ],
    [
        ["q", 88, 146],
        ["r", 34, 124],
        ["s", 96, 564],
        [100, 12, "t"],
    ]
]

t = ""

s = len(A)        # 5 matrices
m = len(A[0])     # 4 rows
n = len(A[0][0])  # 3 columns
```

Output:

```
0 A[0][0][0]=a
1 A[0][0][1]=55
2 A[0][0][2]=146
3 A[0][1][0]=b
4 A[0][1][1]=34
5 A[0][1][2]=124
6 A[0][2][0]=c
7 A[0][2][1]=96
8 A[0][2][2]=564
9 A[0][3][0]=100
10 A[0][3][1]=12
11 A[0][3][2]=d
12 A[1][0][0]=e
13 A[1][0][1]=88
14 A[1][0][2]=146
15 A[1][1][0]=f
16 A[1][1][1]=34
17 A[1][1][2]=124
18 A[1][2][0]=g
19 A[1][2][1]=96
20 A[1][2][2]=564
21 A[1][3][0]=100
22 A[1][3][1]=12
23 A[1][3][2]=h
24 A[2][0][0]=i
25 A[2][0][1]=88
26 A[2][0][2]=146
27 A[2][1][0]=j
28 A[2][1][1]=34
29 A[2][1][2]=124
30 A[2][2][0]=k
31 A[2][2][1]=96
32 A[2][2][2]=564
33 A[2][3][0]=100
34 A[2][3][1]=12
```

```
i = 0
j = 0
d = 0
k = 0

q = n * m * s

for v in range(q):
    k = v % (m * n)
    j = v % n

    if v != 0 and j == 0:
        i += 1
    if v != 0 and k == 0:
        i = 0
        d += 1

    t += str(v) + " A[" + str(d) + \
         "][" + str(i) + "][" + str(j) + "]="

    t += str(A[d][i][j]) + "\n"

print(t)
```

```
35 A[2][3][2]=k
36 A[3][0][0]=m
37 A[3][0][1]=88
38 A[3][0][2]=146
39 A[3][1][0]=n
40 A[3][1][1]=34
41 A[3][1][2]=124
42 A[3][2][0]=o
43 A[3][2][1]=96
44 A[3][2][2]=564
45 A[3][3][0]=100
46 A[3][3][1]=12
47 A[3][3][2]=p
48 A[4][0][0]=q
49 A[4][0][1]=88
50 A[4][0][2]=146
51 A[4][1][0]=r
52 A[4][1][1]=34
53 A[4][1][2]=124
54 A[4][2][0]=s
55 A[4][2][1]=96
56 A[4][2][2]=564
57 A[4][3][0]=100
58 A[4][3][1]=12
59 A[4][3][2]=t
```

This code defines a multi-dimensional array *A*, consisting of 5 matrices, each containing 4 rows and 3 columns of elements. These elements can be a combination of strings and numbers. The code initializes variables *t*, *s*, *m*, and *n*. Variable *t* will be used to accumulate the results, *s* represents the number of matrices in the array, *m* represents the number of rows in each matrix, and *n* represents the number of columns in each matrix. Subsequently, the code uses nested loops to iterate through the elements of the multi-dimensional array *A*. It calculates the total number of elements *q* in the array, which is the product of the number of matrices, rows, and columns. Inside the loop, the code calculates the current position *k*, *j*, *i*, and *d* based on the current iteration *v*. The variables *i*, *j*, *d*, and *k* represent the current row index, column index, matrix index, and overall index, respectively. The code then appends a string to the variable *t* in the format "v A[d][i][j] = value" for each element in the array, where *v* is the current index, *d* is the current matrix index, *i* is the current row index, *j* is the current column index, and *value* is the value of the corresponding element in the array *A*. Next, the code prints the accumulated string *t* to the console, displaying the index and values of all elements in the multi-dimensional array *A*.

6.1.7 Ex. (92) – Traverse a 2D object with a single *for loop* and integer division

```
# integer division 2D one-for loop
# no if then involved.

A = [
   ["a", 88, 146],
   ["b", 34, 124],
   ["c", 96, 564],
   [100, 12, "d"],
   ]

t = ""
n = len(A)    # rows
m = len(A[0]) # columns

for k in range(n * m):
   j = k % m
   i = k // m

   t += f"{k} A[{i}][{j}]={A[i][j]}\n"

print(t)
```

```
Output:

0 A[0][0]=a
1 A[0][1]=88
2 A[0][2]=146
3 A[1][0]=b
4 A[1][1]=34
5 A[1][2]=124
6 A[2][0]=c
7 A[2][1]=96
8 A[2][2]=564
9 A[3][0]=100
10 A[3][1]=12
11 A[3][2]=d
```

The above source code is designed to perform integer division within a two-dimensional array using a single *for-loop*. It starts with the definition of a two-dimensional array A, containing various values and strings. It also initializes an empty string t to store the output and variables n and m to store the number of rows and columns in array A. Two variables i and j are also initialized. The primary goal of this code is to traverse the entire 2D array A using a single loop and output the indices and values of each element in a formatted string. The *for-loop* iterates from 0 to $n \times m$ (exclusive), where n is the number of rows and m is the number of columns in array A. Inside the loop the variable j is calculated as the remainder of dividing k by m, which determines the column index. The variable i is calculated as the result of dividing k by m, rounded down to the nearest integer, which determines the row index. The code builds a string t that contains the current index k and the corresponding element from the 2D array A using the calculated i and j values. The loop continues to the next iteration. Next, after the loop, the *print*(t); statement is used to display the formatted string t, which contains the indices and corresponding elements from the 2D array A. This code effectively provides a structured way to access and display the elements of a 2D array while using a single loop to traverse it.

6.1.8 Ex. (93) – Traverse a 2D object with a single for loop using arithmetic operators

```
A = [
    ["a", 88, 146],
    ["b", 34, 124],
    ["c", 96, 564],
    [100, 12, "d"],
    ]

t = ""
n = len(A)        # rows
m = len(A[0])     # columns

for k in range(n * m):
    j = k % m
    i = (k - j) // m

    t += str(k)+" A["+str(i)+"]["+str(j)+"]="
    t += str(A[i][j]) + "\n"

print(t)
```

Output:

```
0 A[0][0]=a
1 A[0][1]=88
2 A[0][2]=146
3 A[1][0]=b
4 A[1][1]=34
5 A[1][2]=124
6 A[2][0]=c
7 A[2][1]=96
8 A[2][2]=564
9 A[3][0]=100
10 A[3][1]=12
11 A[3][2]=d
```

This source code defines a 2D array A with a mixture of strings and numbers. It initializes an empty string t and determines the number of rows in the A array (n) and the number of columns (m). The code then uses a *for-loop* to iterate through all elements of the array A using a single counter variable k. Inside the loop, it calculates the row i and column j based on the value of k and the dimensions of the array ($n \times m$). For each element of A, the code appends a line to the t string containing the following information: the value of k, the coordinates in the array A where the value is located ($A[i][j]$), and the actual value itself. A newline character is added to separate each line. The code concludes by printing the contents of string t. However, the novelty here is represented by the way variable i and j are calculated. Note that i and j are used to calculate the row and column indices for traversing a two-dimensional array A. To variable n is assigned the value $len(A)$, which represents the number of rows in the array A. Variable m is assigned the value $len(A[0])$, which represents the number of columns in the array A. The loop iterates over all the elements in the two-dimensional array, and for each iteration a series of events unfold. The counter k is used as a linear index that ranges from 0 to $n \times m-1$. It represents the current position within the flattened representation of the 2D array. Variable j is calculated using the modulo operator "%". It calculates the column index by taking the remainder of k divided by m. This gives the column index within the range $[0, m-1]$. In contrast, variable i is calculated using integer division (operator "//"). It calculates the row index by subtracting j from k and then dividing by m. This effectively calculates the row index based on the linear index k and the column index j. The values of k, i, and j are then used to access the corresponding element in the two-dimensional array

A using $A[i][j]$. The values are concatenated to the string t to display the current index and the corresponding element in the array. This code effectively traverses the 2D array by linearizing the indices i and j and using them to access elements in the array in a row-by-row fashion, regardless of the actual array structure.

6.1.9 Ex. (94) – 3D traversal with one for-loop using only arithmetic operators

```
# 3d scan - one for loop - no if then.

A = [
    [
        ["a", 55, 146],
        ["b", 34, 124],
        ["c", 96, 564],
        [100, 12, "d"],
    ],
    [
        ["e", 88, 146],
        ["f", 34, 124],
        ["g", 96, 564],
        [100, 12, "h"],
    ],
    [
        ["i", 88, 146],
        ["j", 34, 124],
        ["k", 96, 564],
        [100, 12, "k"],
    ],
    [
        ["m", 88, 146],
        ["n", 34, 124],
        ["o", 96, 564],
        [100, 12, "p"],
    ],
    [
        ["q", 88, 146],
        ["r", 34, 124],
        ["s", 96, 564],
        [100, 12, "t"],
    ]
    ]
```

```
Output:

0 A[0][0][0]=a
1 A[0][0][1]=55
2 A[0][0][2]=146
3 A[0][1][0]=b
4 A[0][1][1]=34
5 A[0][1][2]=124
6 A[0][2][0]=c
7 A[0][2][1]=96
8 A[0][2][2]=564
9 A[0][3][0]=100
10 A[0][3][1]=12
11 A[0][3][2]=d
12 A[1][0][0]=e
13 A[1][0][1]=88
14 A[1][0][2]=146
15 A[1][1][0]=f
16 A[1][1][1]=34
17 A[1][1][2]=124
18 A[1][2][0]=g
19 A[1][2][1]=96
20 A[1][2][2]=564
21 A[1][3][0]=100
22 A[1][3][1]=12
23 A[1][3][2]=h
24 A[2][0][0]=i
25 A[2][0][1]=88
26 A[2][0][2]=146
27 A[2][1][0]=j
28 A[2][1][1]=34
29 A[2][1][2]=124
30 A[2][2][0]=k
```

```
t = ""

s = len(A)            # 5 matrices
m = len(A[0])         # 4 rows
n = len(A[0][0])      # 3 columns

i = 0
j = 0
d = 0
k = 0

q = n * m * s

for v in range(q):

    k = v % (m*n)

    j = v % n
    i = (k-j) // n
    d = (v-k) // (m*n)

    t += f"{v} A[{d}][{i}][{j}]="
    t += f"{A[d][i][j]}\n"

print(t)
```

```
31 A[2][2][1]=96
32 A[2][2][2]=564
33 A[2][3][0]=100
34 A[2][3][1]=12
35 A[2][3][2]=k
36 A[3][0][0]=m
37 A[3][0][1]=88
38 A[3][0][2]=146
39 A[3][1][0]=n
40 A[3][1][1]=34
41 A[3][1][2]=124
42 A[3][2][0]=o
43 A[3][2][1]=96
44 A[3][2][2]=564
45 A[3][3][0]=100
46 A[3][3][1]=12
47 A[3][3][2]=p
48 A[4][0][0]=q
49 A[4][0][1]=88
50 A[4][0][2]=146
51 A[4][1][0]=r
52 A[4][1][1]=34
53 A[4][1][2]=124
54 A[4][2][0]=s
55 A[4][2][1]=96
56 A[4][2][2]=564
57 A[4][3][0]=100
58 A[4][3][1]=12
59 A[4][3][2]=t
```

In this example, a 3D array A is defined, which represents a multi-dimensional structure containing strings and numbers. The code then initializes several variables, including t (a string for storing the result), s (the number of matrices or "layers" in A), m (the number of rows in each matrix), n (the number of columns in each matrix), and i, j, d, and k as iteration and indexing variables. A loop runs from 0 to q, where q is calculated as the product of n (number of columns), m (number of rows), and s (number of matrices), which effectively iterates through all elements in the 3D array A. Within the loop, the code calculates k as the modulo of v divided by the product of m and n. Variable j is calculated as the modulo of v divided by n, i is calculated as $(k - j)$ divided by n, and d is calculated as $(v - k)$ divided by the product of m and n. These calculations help determine the current position within the 3D array A. The code appends information to the t string for each iteration, showing the current index v and the corresponding value in the A array at the position $[d][i][j]$. A line break is also added to separate the entries. At the end of the loop, the code prints the contents of the t string. Thus, this code iterates over the entire 3D array A and prints the indices d, i, j, and the corresponding element

from the array, effectively displaying the entire content of the 3D array with their indices. Nevertheless, the novelty here is represented by the way variables i, j, and d are computed in order to iterate over the elements of the 3D array A:

$$k = v\%(m \times n)$$

$$j = v\%n$$

$$i = \frac{(k - j)}{n}$$

$$d = \frac{(v - k)}{(m \times n)}$$

Namely, variable k represents the position within a matrix (subarray), and variable i is calculated as $(k - j)$ / n (integer division by using the Python operator: "//"), which is the result of integer division between $k - j$ and the number of columns n. Thus, it calculates the row index within the current matrix. Variable j is calculated as the remainder of v divided by the number of columns n ($j = v \% n$). This gives us the column index within the current matrix. Variable d is calculated as $(v - k)$ / $(m \times n)$, which is the result of integer division between $(v - k)$ and the total number of elements in a matrix $(m \times n)$. Thus, it calculates the index of the current matrix in the 3D array.

Matrix Operations

7

Matrix operations are fundamental mathematical operations used in various fields such as linear algebra, physics, computer science, engineering, and more [1]. Matrices are structured arrangements of numbers or symbols in rows and columns, and these operations allow us to manipulate and analyze data efficiently. They play a crucial role in solving systems of linear equations, transformations, and data analysis. In this context, matrix operations encompass a wide range of mathematical processes, including addition, subtraction, multiplication, inversion, and transposition. Each operation serves a specific purpose and can be applied to matrices of different sizes and dimensions. These operations are not only essential in theoretical mathematics but also have practical applications in computer graphics, machine learning, quantum mechanics, and many other areas. Understanding matrix operations is essential for anyone working with data, whether in scientific research, engineering, or data science. These basic operations provide powerful tools to manipulate, transform, and extract valuable insights from structured data, making them a cornerstone of modern mathematics and science. In the following examples, we will explore various matrix operations and their applications in more detail.

P. A. Gagniuc, *Coding Examples from Simple to Complex*, Synthesis Lectures
on Computer Science, https://doi.org/10.1007/978-3-031-53812-4_7

7.1.1 Ex. (95) – How many 1's in matrix

```
# how many 1's in matrix.

a = [[1, 1, 0, 0, 0],
     [0, 1, 0, 0, 1],
     [1, 0, 0, 1, 1],
     [0, 0, 0, 0, 0],
     [1, 0, 1, 0, 1]]

n = len(a)
m = len(a[0])
k = 0

for i in range(n):
    for j in range(m):
        if a[i][j] == 1:
            k += 1

print(k)
```

Output:

```
10
```

This example is designed to count how many times the value "1" appears in a given matrix. The code begins by defining a matrix a using a two-dimensional array. This matrix consists of rows and columns and contains various values, including 1's and 0's. Next, the code initializes three variables: n, m, and k. Variable n is set to the number of rows in the matrix a (in this case, there are 5 rows). Variable m is set to the number of columns in the matrix a (in this case, there are 5 columns). Also, variable k is initialized to zero and will be used to count the number of "1"s in the matrix. The code then enters a nested loop structure, using *for-loops* to iterate through each element of the matrix. The outer loop, controlled using variable i, iterates through the rows from 0 to $n-1$. The inner loop, is controlled by variable j, that iterates through the columns from 0 to $m-1$. Inside the innermost part of the loop, the code checks if the current element of the matrix, accessed as $a[i][j]$, is equal to "1". If it is, the k variable is incremented by 1. Once both loops complete their iterations, the k variable contains the count of "1"s in the matrix. The code then prints the value of k to the output, which represents the total count of "1"s in the matrix. Thus, the primary purpose of this code is to determine the total count of "1"s in the given matrix a.

7.1.2 Ex. (96) – Sum all values from matrix elements

```
m = [
    [2, 4, 6],
    [3, 5, 6],
    [3, 5, 4]
    ]

r = 0
for i in range(len(m)):
    for j in range(len(m[i])):
        r += m[i][j]

print(r)
```

Output:

```
38
```

This code defines a two-dimensional array *m*, representing a matrix with three rows and three columns. Each element of the matrix contains numeric values. The code then proceeds to calculate the sum of all the elements within this matrix and stores the result in the variable *r*. It uses nested for loops to traverse the entire matrix. The outer loop, controlled by the variable *i*, iterates through the rows of the matrix. In this specific code, it will go through three rows since *len*(*m*) is 3. The inner loop, controlled by the variable *j*, iterates through the elements within each row. In this case, it iterates through the three elements within each row using *len*(*m*[*i*]). For each element in the matrix, the value of that element is added to the running total stored in the variable *r*. This accumulation of values continues as the loops iterate through all the elements in the matrix. Therefore, the script prints out the sum *r* after the loops have processed the entire matrix. This code essentially calculates the sum of all elements in the 3×3 matrix and then displays that total.

7.1.3 Ex. (97) – Show matrix content

```
m = [
    [2, 4, 6],
    [3, 5, 6],
    [3, 5, 4]
    ]

r = ""
for i in range(len(m)):
    r += "\n"
    for j in range(len(m[i])):
        r += str(m[i][j]) + " "

print(r)
```

Output:

```
2 4 6
3 5 6
3 5 4
```

The source code example from above defines a two-dimensional array *m* that contains numeric values organized in rows and columns. The array *m* consists of three rows, and

each row contains three numerical elements. Next, the code initializes an empty string variable *r* which will be used to accumulate the output. The code then enters a nested loop structure using for loops. The outer loop iterates through the rows of the array *m*, and the inner loop iterates through the elements within each row. For each element in the 2D array, the code appends its value to the string *r*, followed by a space, to format the output. After each row is processed in the inner loop, a newline character is added to the string *r* to start a new line for the next row. At the end, the *print* function is called, which typically sends the string *r* to the standard output or console, displaying the entire 2D array in a neatly formatted manner.

7.1.4 Ex. (98) – Multiplication involving a scalar and a matrix.

```
m = [
    [2, 4, 6],
    [3, 5, 6],
    [3, 5, 4]
    ]

s = 3  # scalar

for i in range(len(m)):
    for j in range(len(m[i])):
        m[i][j] = s * m[i][j]
        # m[i][j] *= s

print(m)
```

Output:

```
6,12,18,9,15,18,9,15,12
```

Here, a two-dimensional array *m* is defined. This array contains three subarrays, and each represents a row of a matrix. The matrix *m* holds integer values arranged in a 3 × 3 grid. Next, a scalar variable *s* is set to the value 3. This scalar value will be used to perform scalar multiplication on each element of the matrix *m*. The code enters a nested loop structure. The outer loop iterates over the rows of the matrix *m*, and the inner loop iterates over the elements within each row. This loop structure ensures that every element in the 3 × 3 matrix will be accessed and processed. Within the loop, each element $m[i][j]$ is multiplied by the scalar value *s*, effectively scaling the value of that element. The result of this scalar multiplication overwrites the original value of $m[i][j]$, updating the matrix *m* with the scaled values. Alternatively, the code contains a comment showing a more concise way of achieving the same result using the compound assignment operator "*=". This operator multiplies the value of $m[i][j]$ by *s* and assigns the result back to $m[i][j]$ in a single step. Next, the code ends with a *print* statement, which displays the modified matrix *m* after the scalar multiplication.

7.1.5 Ex. (99) – Sum all values from the rows of the matrix

```
m = [
    [2, 4, 4],
    [3, 5, 6],
    [3, 5, 4]
    ]

r = []

for i in range(len(m)):
    # Initialize r[i] to 0.
    r.append(0)
    for j in range(len(m[i])):
        r[i] += m[i][j]

print(r)
```

Output:

```
10,14,12
```

In this example code, there is an array m defined as a 3×3 matrix, where each element contains integer values. Additionally, there is an empty array r defined, which will be used to store the sum of elements in each row of the matrix m. The code proceeds with two nested loops. The outer loop iterates through the rows of the matrix m. Inside the outer loop, a variable $r[i]$ is initialized to 0 for each row i. This variable will store the sum of elements in row i of the matrix. The inner loop iterates through the elements within each row of the matrix. Within this loop, the code calculates the sum of elements in row i by accumulating the values of $m[i][j]$ into the $r[i]$ variable. The result is an array r where each element $r[i]$ represents the sum of the elements in the corresponding row of matrix m. A comment in the code shows an alternative for a multiplication case, using the "*=" operator, however in this case, using "+=" is appropriate since it adds up the values to calculate the sum (as intended). Next, the code concludes with a *print* statement to display the calculated sums for each row of the matrix m. This output will show the sum of elements in each row as an array r.

7.1.6 Ex. (100) – Sum all values from the colums of the matrix

```
m = [
    [2, 4, 4],
    [3, 5, 6],
    [3, 5, 4]
    ]

c = []

for i in range(len(m)):
    for j in range(len(m[i])):
        if len(c) <= j:
            c.append(0)
        c[j] += m[i][j]
        # c[j] *= m[i][j]

print(c)
```

Output:

```
8,14,14
```

In the provided code, a two-dimensional array m is defined, which represents a 3×3 matrix. The matrix m contains integer values in its cells. Additionally, there is an empty array c declared. This array will be used to store the column-wise sum of elements from matrix m. It is worth noting that c is empty initially, and its length will be determined based on the number of columns in the matrix m. The code then enters a nested loop structure. The outer loop iterates over the rows of the matrix m, and the inner loop iterates over the elements within each row. This loop structure ensures that every element in the 3×3 matrix will be accessed and processed. Within the loop, the code checks whether an element in the c array corresponding to the current column (j) exists. If it does not exist ($len(c) <= j$), it initializes that element to 0. This step is necessary to ensure that the array c has the same number of elements as the number of columns in m. Next, the code adds the value of $m[i][j]$ to the corresponding element in array c, effectively performing column-wise summation. The result is stored in the c array, accumulating the sum as it iterates through the rows. Alternatively, the code contains a comment showing a different operation $c[j] *= m[i][j]$, indicating that any operation is possible in this setup. The code ends with a *print* statement that allows the contents of variable c to be visually observed.

7.1.7 Ex. (101) – Find max and min on columns and rows of a matrix

```
m = [
    [2, 4, 4],
    [3, 5, 6],
    [3, 5, 4]
    ]

a = [
    [0, 0, 0],
    [0, 0, 0],
    [0, 0, 0],
    [0, 0, 0]
    ]

for i in range(len(m)):
    for j in range(len(m[i])):

        if not a[2][j]:
            a[2][j] = 10000
        if not a[3][i]:
            a[3][i] = 10000

        if a[0][j] < m[i][j]:
            a[0][j] = m[i][j]
        if a[1][i] < m[i][j]:
            a[1][i] = m[i][j]

        if a[2][j] > m[i][j]:
            a[2][j] = m[i][j]
        if a[3][i] > m[i][j]:
            a[3][i] = m[i][j]

print('C Max =', a[0])
print('R Max =', a[1])
print('C Min =', a[2])
print('R Min =', a[3])
```

Output:

```
C Max = 3,5,6
R Max = 4,6,5
C Min = 2,4,4
R Min = 2,3,3
```

```
# or an optimised version:


m = [
    [2, 4, 4],
    [3, 5, 6],
    [3, 5, 4]
    ]

a = [
    [0, 0, 0],
    [0, 0, 0],
    [10000, 10000, 10000],
    [10000, 10000, 10000]
    ]

for i in range(len(m)):
    for j in range(len(m[i])):

        a[0][j] = max(a[0][j], m[i][j])
        a[1][i] = max(a[1][i], m[i][j])

        a[2][j] = min(a[2][j], m[i][j])
        a[3][i] = min(a[3][i], m[i][j])

print('C Max =', a[0])
print('R Max =', a[1])
print('C Min =', a[2])
print('R Min =', a[3])
```

In the first example, two 2D arrays, *m* and *a*, are defined. The *m* array represents a 3 × 3 matrix with integer values, and the *a* array represents a 4 × 3 matrix filled with initial values. The code then enters a nested loop structure using two for loops. The outer loop iterates over the rows of the *m* matrix, while the inner loop iterates over the elements within each row. This nested loop structure ensures that every element in the 3 × 3 matrix *m* will be accessed and processed. Inside the loop, there are several conditional statements and assignments. The code checks whether specific elements in the matrix *a* are zero. If they are, it assigns the value 10,000 to these elements. This operation effectively initializes matrix *a* with high values. There are two sets of conditional statements. The first set ($a[0][j]$ and $a[1][i]$) is designed to find the maximum value along each column and row of the *m* matrix. The second set ($a[2][j]$ and $a[3][i]$) is used to find the minimum value along each column and row of the *m* matrix. The code uses conditional comparisons to update the values in the *a* matrix based on the conditions mentioned above. Next, the code outputs the results using the *print* statement, which displays the following statistics: (1) "C Max" represents the maximum value for each column of the *m* matrix. (2) "R Max" represents the maximum value for each row of matrix *m*. (3) "C Min" represents the minimum value for each column of matrix *m*. (4) "R Min" represents the minimum

value for each row of *m*. A second version of the code, below the first example, simplifies the implementation, by using the *max* and *min* built-in functions.

7.1.8 Ex. (102) – Multiply all values from the columns / rows and store them in array

```
m = [
    [2, 4, 4],
    [3, 5, 6],
    [3, 5, 4]
    ]

# Initialize a with
# default values.

a = [[1 for _ in range(len(m[0]))], \
     [1 for _ in range(len(m))]]

for i in range(len(m)):
    for j in range(len(m[i])):
        # Check if the element
        # exists, if not
        # initialize to 1.

        # In Python, this check
        # is not necessary since
        # we have initialized all
        # elements to 1.

        # if not a[0][j]: a[0][j] = 1
        # if not a[1][i]: a[1][i] = 1

        a[0][j] *= m[i][j]
        a[1][i] *= m[i][j]

        # The commented out code
        # is an alternative
        # calculation, not used in
        # the active code.

        # if not a[0][j]: a[0][j] = 0
        # if not a[1][i]: a[1][i] = 0
        # a[0][j] += m[i][j]
        # a[1][i] += m[i][j]

print('C =', a[0])
print('R =', a[1])


# or another version:


m = [
    [2, 4, 4],
    [3, 5, 6],
    [3, 5, 4]
    ]
```

Output:

```
C = 18,100,96
R = 32,90,60
```

```
a = [[1] * len(m[0]), [1] * len(m)]

for i in range(len(m)):
    for j in range(len(m[i])):
        a[0][j] *= m[i][j]
        a[1][i] *= m[i][j]

print('C = ' + str(a[0]))
print('R = ' + str(a[1]))
```

In the first code, a two-dimensional array *m* is defined, just like in the previous example. This array contains three subarrays, representing a 3 × 3 matrix, with integer values. Additionally, two empty arrays *a* are created and initialized a with default values. These arrays are intended to store the product of elements in the matrix *m* along rows (R) and columns (C). The code then enters a nested loop structure, similar to the previous example, where the outer loop iterates over the rows of the matrix *m*, and the inner loop iterates over the elements within each row. Within the loop, there is a commented conditional logic to handle the initialization of elements in the arrays *a*. If the specific element in *a*[0][*j*] or *a*[1][*i*] does not exist (i.e., it is undefined), it is initialized to 1. This condition ensures that if the element is accessed for the first time, it is set to 1. Nevertheless, the comments section is only one of the solutions. That is, because *a* is initialized with values, the commented section is not needed. Next, each element in *a*[0][*j*] and *a*[1][*i*] is multiplied by the corresponding element from the matrix *m*[*i*][*j*]. This step calculates the product of elements along columns and rows, as the code progresses through the matrix *m*. The code also includes commented-out alternative approaches using addition ("+=") instead of multiplication ("*="), or initializing with 0 and then adding the matrix element, which would result in calculating the sum along columns and rows. Next, the code displays the product along columns as "C" and the product along rows as "R", followed by the respective arrays containing these results.

7.1.9 Ex. (103) – Sum all values from the right diagonal of the square matrix

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

d = 0

n = len(a)
m = len(a[0])

i = 0
while i < n:
    j = 0
    while j < m:
        d += a[i][m-j-1]
        print(a[i][m-j-1])
        i += 1
        j += 1

print('---\n' + str(d))


# or a second version:


a = [
    [1, 2, 3],
    [4, 5, 6],
    [7, 8, 9]
    ]

d = 0
n = len(a)
m = len(a[0])
i = 0

for j in range(m):
    d += a[i][m-j-1]
    print(a[i][m-j-1])
    i += 1

print('---\n' + str(d))
```

Output:

```
3
5
7
---
15
```

In this case, a two-dimensional array *a* is defined, which represents a 3 × 3 matrix containing integer values. The code aims to perform specific operations on this matrix and calculate a sum, denoted by the variable *d*. Initially, a variable *d* is declared and initialized with the value 0. This variable will be used to accumulate the sum of specific elements in the matrix. The code then retrieves the dimensions of the matrix *a*. It uses the length property to determine the number of rows (*n*) and the number of columns (*m*) in the matrix. Subsequently, the code enters a nested loop structure. The outer loop iterates over the rows of the matrix using the variable *i*, ranging from 0 to *n*-1. The inner loop, controlled by *j*, iterates over the columns of the matrix, ranging from 0 to *m*-1. Within the inner loop, the code calculates the column index to access elements in reverse order from right to left within each row using the expression *m-j*-1. The element *a*[*i*][*m-j*-1] is accessed, and its value is added to the variable *d* to accumulate the sum. Additionally, the code shows a *print* statement within the inner loop, meant to output each value that participates to the summation. Thus, the statement *print*(*a*[*i*][*m-j*-1]) displays the values of the matrix elements accessed during the iteration. At the end of the code, there is another *print* statement that displays a line containing three hyphens "---" followed by the value of *d*. This is intended to show the sum of the selected elements in the matrix. Note that *i*+=1 is meant to increment the *i* counter in the inner loop, which effectively makes the outer loop useless. That is, the counter *i* reaches the value of *n* in the inner loop, because the example deals with a square matrix where the columns and rows are equal. Thus, this code can be reduced to a single loop (see the second version below the first).

7.1.10 Ex. (104) – Sum all values from the left diagonal of the square matrix

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

d = 0
n = len(a)
m = len(a[0])
i = 0

for j in range(m):
    d += a[i][j]
    print(a[i][j])
    i += 1

print('---\n' + str(d))
```

Output:

```
1
5
9
---
15
```

Above, a two-dimensional array *a* is defined. This array represents a 3 × 3 matrix, with each subarray corresponding to a row of the matrix. Additionally, a variable *d* is initialized with the value 0, which will be used to accumulate the sum of all elements in the matrix. The code then proceeds to determine the dimensions of the matrix. Variable *n* is set to the number of rows (which is 3 in this case), and *m* is set to the number of columns (also 3). The loop, controlled by variable *j*, iterates over the columns of the matrix. Within this loop, the code calculates the sum of the elements by adding the value of each element *a*[*i*][*j*] to the accumulator *d*. The value of the element *a*[*i*][*j*] is also printed in the console using the *print* function. Additionally, inside this loop there is an *i+=1* statement, which increments the value of *i* in order to avoid the use of a nested loop. After processing all elements of the matrix, the script ends by printing a separator line ("---\n") followed by the accumulated sum *d* which represents the sum of all elements in the left diagonal of the matrix.

7.1.11 Ex. (105) – Sum all values from the left and right diagonal of a square matrix

```
a = [
    [1, 2, 3, 4],
    [5, 6, 7, 8],
    [9, 0, 1, 2],
    [3, 4, 5, 6]
    ]

ld = 0
rd = 0

n = len(a)
m = len(a[0])

i = 0

for j in range(m):
    ld += a[i][j]
    rd += a[i][m-j-1]
    i += 1

print('L=' + str(ld) + '|R=' + str(rd))
```

Output:

```
L=14|R=14
```

In this Python code, we are working with a two-dimensional array *a*, which represents a 4 × 4 grid. This array contains four subarrays, each of which corresponds to a row of the matrix. Additionally, two variables are defined: *ld* (for left diagonal) and *rd* (for right diagonal), both initially set to 0. These variables will be used to accumulate the sums of the values along the left and right diagonals of the matrix, respectively. The code proceeds with setting the variables *n* and *m*, where *n* represents the number of rows in the matrix (which is 4 in this case), and *m* represents the number of columns (also 4). Moreover, a counter variable *i* is also initialized to zero. Next, a *for-loop* structure is used. Within the loop, two actions are performed. The *ld* (left diagonal) variable is updated by adding the value of the current element *a*[*i*][*j*]. This step accumulates the sum of values along the left diagonal of the matrix. The *rd* (right diagonal) is updated by adding the value of the element at *a*[*i*][*m-j*-1]. This step accumulates the sum of values along the right diagonal of the matrix. The *m-j*-1 expression accesses the elements in reverse order along each row, effectively giving the values on the right diagonal. Notably, the code contains the *i+=1* statement in the main loop, which, as in the previous examples, is meant to eliminate the use of a nested loop. The code concludes with a *print* statement that displays the results. The sums of the left and right diagonals, stored in the *ld* and *rd* variables, respectively, are presented in the following format: "L=*ld*|R=*rd*."

7.1.12 Ex. (106) – Sum all values from the left and right diagonal by using conditions

```
# Sum on principal and secondary diagonals.

a = [
    [1, 2, 3, 4],
    [5, 6, 7, 8],
    [9, 0, 1, 2],
    [3, 4, 5, 6]
    ]

d = [0, 0]

n = len(a)
m = len(a[0])

for i in range(n):
    for j in range(m):
        if i == j:
            d[0] += a[i][j]
        if i + j == n - 1:
            d[1] += a[i][j]

print('L=' + str(d[0]) + '|R=' + str(d[1]))
```

```
Output:

L=14|R=14
```

The previous code contained an optimal way to sum the values found on the diagonals of a matrix, without using conditions. However, what *if* conditional branching is used, what would the example normally look like? In this new version, we are working with a two-dimensional array a, which represents a square matrix with dimensions 4×4. The array d is initialized as an empty array, and it is used to store the sums of the elements along the principal diagonal and the secondary diagonal of the matrix a. The principal diagonal of a square matrix consists of the elements where the row and column indices are the same ($i = j$), and the secondary diagonal consists of the elements where the sum of the row and column indices is equal to one less than the number of rows ($i + j = n - 1$). The variable n is set to the number of rows in the matrix a, and m is set to the number of columns in the matrix a. In this case, both n and m are set to 4 since the matrix a is 4×4. The code enters a nested loop structure. The outer loop iterates over the rows of the matrix a, and the inner loop iterates over the columns of the matrix a. Within this loop, there are conditional statements that check if the current element is part of the principal diagonal or the secondary diagonal. If i (the current row index) is equal to j (the current column index), the element belongs to the principal diagonal, and its value is added to $d[0]$. This operation accumulates the sum of the elements on the principal diagonal. If $(i + j)$ is equal to $(n - 1)$, the element belongs to the secondary diagonal, and its value is added to $d[1]$. This operation accumulates the sum of the elements on the secondary diagonal. Next, the code prints a string that represents the results. It displays the sum of the elements on the principal diagonal (L) and the sum of the elements on the secondary diagonal (R) in the format "L=value|R=value."

7.1.13 Ex. (107) – Show bottom – left part of the matrix

```
a = [
    [1, 2, 3, 4],
    [5, 6, 7, 8],
    [9, 0, 1, 2],
    [3, 4, 5, 6]
    ]

d = []
n = len(a)

r = ''

for i in range(n):
    d.append([])
    for j in range(i+1):
        d[i].append(a[i][j])
        r += str(d[i][j]) + ' '
    r += '\n'

print(r)
```

```
Output:

1
5 6
9 0 1
3 4 5 6
```

```
'''

1 - - -
5 6 - -
9 0 1 -
3 4 5 6

'''
```

Other operations may involve selective parts of the matrix. Thus the above Python code defines a 2D array *a* containing a grid of numbers. It then initializes an empty array *d* and extracts the dimensions of the array *a*, storing the number of rows in *n* and the number of columns in *m*. Next, a string variable *r* is also declared and initialized as an empty string. The code proceeds to loop through the rows of the array *a* by using a *for-loop*, with the index variable *i*. Within this loop, another nested loop runs with index variable *j* to iterate through the columns. During each iteration, the value from the array *a* at the *i* row and *j* column is stored in the *d* array at the same position. Additionally, the value is appended to the *r* string, separated by a space. After each row of *d* is processed, a newline character is appended to the *r* string to separate the rows. Next, the code prints the contents of the *r* string using the *print* function. Overall, this code generates a new 2D array *d* that contains a triangular subset of the original array *a*, and *r* holds a string representation of this subset.

7.1.14 Ex. (108) – Show bottom – left part of the matrix and flip horizontal

```
a = [
    [1, 2, 3, 4],
    [5, 6, 7, 8],
    [9, 0, 1, 2],
    [3, 4, 5, 6]
    ]

d = []
n = len(a)
m = len(a[0])
r = ''

for i in range(n):
    d.append([])
    for j in range(i + 1):
        d[i].append(a[i][i-j])
        r += str(d[i][j]) + ' '
    r += '\n'

print(r)
```

```
Output:

1
6 5
1 0 9
6 5 4 3

'''

1 - - -
5 6 - -
9 0 1 -
3 4 5 6

'''
```

In this example, we start with a 2D array *a* that contains a grid of numbers. The code then initializes an empty array *d* and extracts the dimensions of the array *a*, storing the number of rows in *n* and the number of columns in *m*. The variable *r* is declared and initialized as an empty string. The code uses a *for-loop* to iterate through the rows of the *a* array using the index variable *i*. Within this loop, there is a nested loop that iterates through the columns. During each iteration, the value from the array *a* at the *i* row and (*i-j*) column is stored in the *d* array at the same position. The value is also appended to the *r* string, separated by a space. After each row of *d* is processed, a newline character is appended to the *r* string to separate the rows. As in the previous example, the code prints the contents of the *r* string using the *print* function. Please note that this source code example generates a new 2D array *d* that contains a triangular subset of the original array *a*, but this time the values are selected from the opposite diagonal of *a*, and *r* holds a string representation of this subset.

7.1.15 Ex. (109) – Show top – right part of the matrix

```
a = [
    [1, 2, 3, 4],
    [5, 6, 7, 8],
    [9, 0, 1, 2],
    [3, 4, 5, 6]
    ]

d = []
n = len(a)
m = len(a[0])

r = ''

for i in range(n):

    # Initializing the inner
    # list with None values.

    d.append([None] * m)
    for j in range(i, m):
        d[i][j] = a[i][j]
        r += str(d[i][j]) + ' '
    r += '\n'

print(r)
```

Output:

```
1 2 3 4
6 7 8
1 2
6
```

```
'''

1 2 3 4
- 6 7 8
- - 1 2
- - - 6

'''
```

This Python example operates on a 2D array *a* and produces a different result compared to the previous code. The code begins by defining a 2D array *a* with four rows and four columns. It also initializes an empty array *d*, and it determines the number of rows *n* and columns *m* in the array *a*. Just as before, a string variable *r* is declared and initialized as an empty string. The code then uses a nested loop structure. The outer loop, controlled by the index variable *i*, iterates through the rows of the array *a* from 0 to *n*-1. Within the outer loop, a nested loop (controlled by the index variable *j*) starts from *i* and goes up to the last column *m* (exclusive). During each iteration, the value from the array *a* at the *i* row and *j* column is stored in the *d* array at the same position. Additionally, the value is appended to the *r* string, separated by a space. After each column is processed within a row, a newline character is added to the *r* string to separate the rows. In the next stage, the code prints the contents of the *r* string. Thus, this code generates a new 2D array *d* containing a right triangular subset of the original array *a*, and *r* holds a string representation of this subset.

7.1.16 Ex. (110) – Show top – right part of the matrix and flip 90 degrees left

```
a = [
    [1, 2, 3, 4],
    [5, 6, 7, 8],
    [9, 0, 1, 2],
    [3, 4, 5, 6]
    ]

d = []
n = len(a)
r = ''

for i in range(n):
    d.append([])
    for j in range(i + 1):
        d[i].append(a[i-j][i])
        r += str(d[i][j]) + ' '
    r += '\n'

print(r)
```

Output:

```
1
6 2
1 7 3
6 2 8 4
```

```
'''


1 2 3 4
- 6 7 8
- - 1 2
- - - 6

'''
```

In this code example all variables are declared just as in the previous example. The code proceeds to loop through the rows of the array using a *for-loop*, just as done previously. However, the key difference is in how the values are obtained. In this case, the values from the array *a* are taken from different positions. Specifically, the value from *a* at the row *i-j* and column *i* is stored in the *d* array at the same position *i*. The rest of the code is similar to the previous example. The values are appended to the *r* string with a space between them, and a newline character is added after each row is processed. Just like the previous example, the code prints the contents of the *r* string using the *print* function. As perhaps it can be observed, this code generates a new 2D array *d* that contains a triangular subset of the original array *a*, but this time the values are selected differently, resulting in a different subset, stored in variable *r*.

7.1.17 Ex. (111) – Show top – right, flip 90 degrees left and flip horizontally

```
a = [
    [1, 2, 3, 4],
    [5, 6, 7, 8],
    [9, 0, 1, 2],
    [3, 4, 5, 6]
    ]

d = []
n = len(a)
r = ''

for i in range(n):
    d.append([])
    for j in range(i + 1):
        d[i].append(a[j][i])
        r += str(d[i][j]) + ' '
    r += '\n'

print(r)
```

Output:

```
1
2 6
3 7 1
4 8 2 6
```

```
'''

1 2 3 4
- 6 7 8
- - 1 2
- - - 6

'''
```

This Python code is similar to the previous one but with a different purpose. It starts by defining a 2D array *a* that contains a grid of numbers. Then, it initializes an empty array *d* and retrieves the number of rows in *n* and the number of columns in *m* from the array *a*. Variable *r* is also declared and initialized as an empty string. The code proceeds to loop through the rows of the array *a* using a *for-loop* with the index variable *i*. Within this loop, there is another nested loop that iterates through the columns with the index variable *j*. During each iteration, the value from the array *a* at the *j* row and *i* column (note the reversed indices compared to the previous code) is stored in array *d* at the same position. Additionally, the value is appended to the *r* string, separated by a space. After each row of *d* is processed, a newline character is appended to the *r* string to separate the rows. Just like in the previous code, the *print* function is used to display the contents of the *r* string. In conclusion, this code generates a new 2D array *d* that contains a transposed version of the original array *a*, and *r* holds a string representation of this transposed array.

7.1.18 Ex. (112) – Secondary diagonal scan (right)

```
a = [
    [1, 2, 3, 4],
    [5, 6, 7, 8],
    [9, 0, 1, 2],
    [3, 4, 5, 6]
    ]

d = []
n = len(a)
r = ''

for i in range(n):
    d.append([])
    for j in range(i + 1):
        d[i].append(a[j][i-j])
        r += str(d[i][j]) + ' '
    r += '\n'

print(r)
```

```
Output:

1
2 5
3 6 9
4 7 0 3
```

```
'''

1 2 3 4
5 6 7 -
9 0 - -
3 - - -

'''
```

This code generates a new 2D array *d* that contains a transformed subset of the original *a* array, and *r* holds a string representation of this transformed subset. It operates on a 2D array *a* and performs a different transformation compared to the previous code. The key difference here is that the value from *a* is not extracted directly based on row and column indices as in the previous code. Instead, it uses *a*[*j*][*i*-*j*] to obtain values from *a* by a different pattern. For each iteration, the value from *a* obtained using *a*[*j*][*i*-*j*] is stored in the *d* array at the same position (*d*[*i*][*j*]), and this value is appended to the *r* string, separated by a space. A newline character is also appended to *r* after each row of *d* is processed.

7.1.19 Ex. (113) – Secondary diagonal scan (left)

```
a = [
    [1, 2, 3, 4],
    [5, 6, 7, 8],
    [9, 0, 1, 2],
    [3, 4, 5, 6]
    ]

# Initialize the empty list d,
# and define n and m.

d = []
n = len(a)
m = len(a[0])
r = ''

for i in range(n):
    d.append([])
    for j in range(i + 1):
        d[i].append(a[i - j][n-1-j])
        r += str(d[i][j]) + ' '
    r += '\n'

print(r)
```

Output:

```
4
8 3
2 7 2
6 1 6 1
```

```
'''

1 2 3 4
- 6 7 8
- - 1 2
- - - 6

'''
```

Here, the example generates a new matrix *d* that contains a triangular subset of the original array *a*, following a different pattern compared to the previous code. The code initializes an empty matrix *d* and extracts the dimensions of the matrix *a*, storing the number of rows and columns in *n* and *m*, respectively. An empty string *r* acting as an accumulator is declared and initialized. The code then enters a loop that iterates through the rows of array *a* using a *for-loop* with the index variable *i*. Within this loop, there is another nested loop with the index variable *j* that iterates through the columns. During each iteration, the code accesses elements from array *a* in a pattern where the row index is based on *i* and *j*, and the column index is derived from *n* and *j*. The value from array *a* at this calculated position is then stored in array *d* at the corresponding *i* and *j* position. The value is also appended to the content of the *r* variable with a space separator. Once the processing of each row of *d* is made, a newline character is added to the *r* string to separate the rows.

7.1.20 Ex. (114) – Show bottom – right and flip horizontally

```
a = [
    [1, 2, 3, 4],
    [5, 6, 7, 8],
    [9, 0, 1, 2],
    [3, 4, 5, 6]
    ]

d = []
n = len(a)
m = len(a[0])

r = ''

for i in range(n):
    d.append([])
    for j in range(i + 1):
        d[i].append(a[i][m-j-1])
        r += str(d[i][j]) + ' '
    r += '\n'

print(r)
```

Output:

```
4
8 7
2 1 0
6 5 4 3
```

```
'''

- - - 4
- - 7 8
- 0 1 2
3 4 5 6

'''
```

This code generates a new 2D array *d* that contains a triangular subset of the original array *a* with the elements reversed within each row. The Python example from above is similar to the previous one, but with a slight modification in the way it populates the array *d* from array *a*. The code then enters a nested loop structure using a *for-loop* to iterate through the rows and columns. During each iteration, the value from the array *a* at the *i* row and *m-j*-1 column is stored in array *d* at the same position. The *m-j*-1 index is used to select elements in reverse order within each row of array *a*, effectively reversing the order of elements in each row.

7.1.21 Ex. (115) – Matrix flip vertical

```
# flip vertical.

a = [
    [1, 2, 3, 4],
    [5, 6, 7, 8],
    [9, 0, 1, 2],
    [3, 4, 5, 6]
    ]

d = []
n = len(a)
m = len(a[0])

r = ''

for i in range(n):

    # Initialize the
    # inner list.

    d.append([0] * m)

    for j in range(m):
        d[i][j] = a[n-i-1][j]
        r += str(d[i][j]) + ' '
    r += '\n'

print(r)
```

Output:

```
3 4 5 6
9 0 1 2
5 6 7 8
1 2 3 4
```

The above code flips the original array *a* vertically to create a new 2D array *d*, and *r* holds a string representation of this flipped array. The implementation, like before, begins with a 2D array *a* containing a grid of numbers. It then initializes an empty array *d* and determines the dimensions of the array *a*. The number of rows is stored in *n*, and the number of columns is stored in *m*. The code proceeds to loop through the rows of array *a* using a *for-loop*, with the index variable *i*. Within this loop, there is another nested loop that iterates through the columns using the index variable *j*. During each iteration, the code assigns a value from array *a* to array *d*, effectively flipping the rows vertically. It takes the *i*-th row from *a* in reverse order (*n*-*i*-1) and assigns it to the *i*-th row of *d*. Additionally, the value is appended to the *r* string, separated by a space. After processing each row of *d*, a newline character is appended to the *r* string to separate the rows. The code then shows the contents of *r*.

7.1.22 Ex. (116) – Matrix flip horizontal

```
# flip horisontal.

a = [
    [1, 2, 3, 4],
    [5, 6, 7, 8],
    [9, 0, 1, 2],
    [3, 4, 5, 6]
    ]

d = []
n = len(a)
m = len(a[0])

r = ''

for i in range(n):
    d.append([])
    for j in range(m):
        d[i].append(a[i][m-j-1])
        r += str(d[i][j]) + ' '
    r += '\n'

print(r)
```

```
Output:

4 3 2 1
8 7 6 5
2 1 0 9
6 5 4 3
```

This code flips an array *a* horizontally and stores the flipped version in the array *d*, with *r* containing a string representation of this flipped array. The code starts with the definition of a 2D array *a*, which contains a grid of numbers. It initializes an empty array *d* and extracts the number of rows (*n*) and columns (*m*) of the array *a*. The code then proceeds to iterate through the rows of the array *a* using a *for-loop* with the index variable *i*. Within this loop, there is another nested loop that iterates through the columns using the index variable *j*. During each iteration, the value from the *a* array at the *i* row and *m-j*-1 column is stored in the *d* array at the same position. This effectively flips the columns of array *a* horizontally. Additionally, the value is appended to the *r* string, separated by a space. After processing each row of *d*, a newline character is appended to the *r* string to separate the rows. The content of the variable is then printed in the output.

7.1.23 Ex. (117) – Sum values from elements of a matrix based on a map

```
a = [
    [2, 4, 6],
    [3, 5, 6],
    [3, 5, 4]
    ]

b = [
    [0, 0, 1],
    [1, 1, 0],
    [0, 0, 1]
    ]

n = len(a)
m = len(a[0])

r = 0

for i in range(n):
    for j in range(m):
        if b[i][j] == 1:
            r += a[i][j]

print(r)
```

Output:

```
18
```

This source code example calculates the sum of the values in matrix *a* where the corresponding elements in the *b* matrix are equal to 1 and stores the result in the *r* variable. Above, the code defines two 2D arrays, *a* and *b*, which represent matrices of numbers. It then calculates the dimensions of the array *a*, storing the number of rows in *n* and the number of columns in *m*. A variable *r* is declared and initialized to zero. The code proceeds to loop through the rows of the array *a* using a *for-loop* with the index variable *i*. Inside this loop, there is a nested loop that iterates through the columns using the index variable *j*. During each iteration, the code checks if the value of *b* at the *i* row and *j* column is equal to 1. If it is, the corresponding value from the array *a* at the same position is added to the *r* variable. Thus, the code prints the value of *r* using the *print* function.

7.1.24 Ex. (118) – Add two matrices into a third

```
a = [
    [2, 4, 6],
    [3, 5, 6],
    [3, 5, 4]
    ]

b = [
    [2, 4, 6],
    [3, 5, 6],
    [3, 5, 4]
    ]

c = []
r = ""

for i in range(len(a)):
    r += "\n"
    c.append([])

    for j in range(len(a[i])):
        c[i].append(a[i][j] + b[i][j])
        r += str(c[i][j]) + " "

print(r)
```

Output:

```
4 8 12
6 10 12
6 10 8
```

```
'''
for substraction:

0 0 0
0 0 0
0 0 0

'''
```

```
# or a second version:


a = [
    [2, 4, 6],
    [3, 5, 6],
    [3, 5, 4]
    ]

b = [
    [2, 4, 6],
    [3, 5, 6],
    [3, 5, 4]
    ]

# Pre-allocate the size of c.

c = [[0] * len(a[0]) for _ in range(len(a))]

r = ""

for i in range(len(a)):
    r += "\n"

    for j in range(len(a[i])):
        c[i][j] = a[i][j] + b[i][j]
        r += str(c[i][j]) + " "

print(r)
```

Here, the first version of the code begins by defining two 2D arrays, *a* and *b*, each containing a grid of numbers. It then initializes an empty array *c* and an empty string *r*. The code uses a nested loop to iterate through the rows and columns of *a* and *b*. The outer loop iterates through the rows of *a* and *b* using the index variable *i*. Within this loop, the code appends a newline character to the *r* string to separate rows and initializes an empty sub-array in *c* at index *i*. The inner loop iterates through the columns of *a* and *b* using the index variable *j*. In each iteration, the code calculates the sum of the corresponding elements in *a* and *b* and stores the result in the *c* array at the same position. The sum is also appended to the *r* string, separated by a space. The code then prints the contents of the *r* string. Thus, this code calculates the element-wise sum of the *a* and *b* arrays and stores the result in the *c* array. The *r* string contains a formatted representation of the resulting array. There is also a second version of this code that uses a preallocated number of elements for *c*, to be immediately accessible in the main nested *for-loops*. Note that in the first version the array *c* was not preallocated, but instead an element was appended when needed (.append).

7.1.25 Ex. (119) – Matrix multiplication with three for loops

```
a = [
    [2, 4, 6],
    [3, 5, 6],
    [3, 5, 4]
    ]

b = [
    [2, 4, 6],
    [3, 5, 6],
    [3, 5, 4]
    ]

c = []
r = ""

for i in range(len(a)):
    r += "\n"
    c.append([])

    for j in range(len(a[i])):
        c[i].append(0)

        for k in range(len(a[i])):
            c[i][j] += a[k][j] * b[i][k]
        r += str(c[i][j]) + " "

print(r)
```

Output:

```
34 58 60
39 67 72
33 57 64
```

In this code implementation, there are three 2D arrays: *a*, *b*, and *c*. The code performs matrix multiplication between *a* and *b* and stores the result in *c*. A string variable *r* is initialized and will be used to store a string representation of the *c* matrix. The code uses nested for loops to iterate through the rows and columns of the *a* and *b* matrices. The outer loop, controlled by the variable *i*, iterates over the rows of a and *b*. For each row, a new row is created in the *c* matrix, and a newline character is added to the *r* string to separate the rows in the string representation. The innermost loop, controlled by variable *k*, is responsible for performing the matrix multiplication. It calculates the value of $c[i][j]$ by summing up the products of elements from *a* and *b* matrices. The result is stored in the *c* matrix at the corresponding position, and the calculated value is appended to the *r* string followed by a space. Once all the iterations are completed, the *r* string contains a string representation of the resulting *c* matrix, which is essentially the product of matrices *a* and *b*.

7.1.26 Ex. (120) – Matrix multiplication with two for loops

```
a = [
    [2, 4, 6],
    [3, 5, 6],
    [3, 5, 4]
    ]

b = [
    [2, 4, 6],
    [3, 5, 6],
    [3, 5, 4]
    ]

i = j = 0
r = ""
c = []

n1 = len(a)
n2 = len(a[0])
q = n1 * n2

c.append([])

for v in range(q):

    j = v % n2

    if j==0 and v!=0 and i<n1 and v!=q:
        i += 1

    c.append([])
    r += "\n"
    c[i].append(0)

    for k in range(len(a[i])):
        c[i][j] += a[k][j] * b[i][k]
        r += str(c[i][j]) + " "

print(r)
```

```
Output:

34 58 60
39 67 72
33 57 64
```

The example above performs matrix multiplication between two 2D arrays *a* and *b*. It calculates the product and stores the result in the *c* array. The *r* string is used to format and print the result. Two 2D arrays, *a* and *b*, are defined, each containing three rows and three columns. Index variables *i* and *j* are initialized to 0. The *r* string is initialized as an empty string, and an empty array *c* is defined to store the result. The dimensions of *a* are determined with *n1* representing the number of rows and *n2* representing the number of columns. A loop is used to iterate from 0 to the total number of elements in *a* (*n1* * *n2*), denoted as *q*. The loop variable is *v*. Within the loop, the code calculates the current column index *j* as *v* % *n2*. If *j* is 0, a new row is started by incrementing *i*, and a new empty sub-array is added to *c*. A newline character is also appended to the *r* string to

separate rows. The code sets $c[i][j]$ to 0 to prepare it for storing the matrix multiplication result. A nested loop iterates through the elements of a and calculates the dot product of the j-th column of a and the i-th row of b. The result is stored in $c[i][j]$. The calculated result, $c[i][j]$, is appended to the r string followed by a space to separate values. After the loop completes, the r string contains the formatted result of the matrix multiplication. The code prints the contents of the r string in the console by using the *print* function.

7.1.27 Ex. (121) – Multiply specific elements of two matrices based on a map

```
a = [
    [2, 4, 6],
    [3, 5, 6],
    [3, 5, 4]
    ]

b = [
    [0, 1, 0],
    [1, 1, 1],
    [0, 1, 0]
    ]

c = [
    [2, 4, 6],
    [3, 5, 6],
    [3, 5, 4]
    ]

n = len(a)
m = len(a[0])
r = ''

for i in range(n):
    r += '\n'
    for j in range(m):
        if b[i][j] == 1:
            c[i][j] = a[i][j] * c[i][j]
        r += str(c[i][j]) + " "

print(r)
```

Output:

```
2 16 6
9 25 36
3 25 4
```

This code performs element-wise multiplication on the *c* array using values from the array *a* based on the *b* array values, and it generates a string *r* containing the updated *c* array with spaces and newlines to format the output. Initially, three 2D arrays *a*, *b*, and *c* are defined. The *a* and *b* arrays contain numeric values, while the *c* array is initially identical to the *a* array. The code initializes variables *n* and *m* with the number of rows and columns in the array *a*. A string variable *r* is also initialized as an empty string. The code then enters a nested loop structure, iterating over the rows (index *i*) and columns (index *j*) of the *a* and *b* arrays. Inside the loop, it checks if the value of *b* at the current position (*i*, *j*) is equal to 1. If it is, it multiplies the corresponding value in the *c* array by the value in the array *a* at the same position. Regardless of whether the multiplication occurs or not, the value from the *c* array at the current position is appended to the *r* string, separated by a space. After all columns in a row are processed, a newline character is added to the *r* string to separate the rows. Next, the code prints the contents of the *r* string.

7.1.28 Ex. (122) – Different operations based on maps

```
# perform different operations between
# the values of the homologous elements
# of two arrays based on a map/model
# (third array).

a = [
    [2, 2, 2, 2, 2],
    [2, 2, 2, 2, 2],
    [2, 2, 2, 2, 2],
    [2, 2, 2, 2, 2],
    [2, 2, 2, 2, 2]
    ]

b = [
    [1, 1, 0, 0, 0],
    [3, 1, 0, 0, 1],
    [1, 3, 0, 1, 1],
    [0, 0, 0, 7, 0],
    [3, 0, 4, 0, 9]
    ]

c = [
    [3, 3, 3, 3, 3],
    [3, 3, 3, 3, 3],
    [3, 3, 3, 3, 3],
    [3, 3, 3, 3, 3],
    [3, 3, 3, 3, 1]
```

Output:

```
5 5 6 6 6
1 5 6 6 5
5 1 6 5 5
6 6 6 # 6
1 6 2 6 2
```

```
    ]

n = len(a)
m = len(a[0])
r = ''

for i in range(n):
    r += '\n'
    for j in range(m):
        if b[i][j] == 0:
            c[i][j] = a[i][j] * c[i][j]
        if b[i][j] == 1:
            c[i][j] = a[i][j] + c[i][j]
        if b[i][j] == 2:
            c[i][j] = a[i][j] - c[i][j]
        if b[i][j] == 3:
            c[i][j] = c[i][j] - a[i][j]
        if b[i][j] == 4:
            c[i][j] = a[i][j] % c[i][j]
        if b[i][j] == 5:
            c[i][j] = a[i][j] / c[i][j]
        if b[i][j] == 6:
            c[i][j] = a[i][j]
        if b[i][j] == 7:
            c[i][j] = '#'
        if b[i][j] == 8:
            pass  # Placeholder for "do stuff"
        if b[i][j] == 9:
            if c[i][j] <= a[i][j]:
                c[i][j] = a[i][j]

        r += str(c[i][j]) + " "

print(r)
```

This code operates on two matrices *a* and *c* based on a specified map/pattern found in matrix *b*, and then stores the results in matrix *c*. It then prints the resulting *c* matrix as a string representation. The code begins by defining three matrices *a*, *b*, and *c*. Variables *a* and *b* contain numeric values, while *c* contains initial values, which are all 3s except for the bottom-right element, which is 1. The code determines the dimensions of the matrices, with *n* representing the number of rows and *m* representing the number of columns. Like many times in the examples from above, it also initializes an empty string *r* for storing the final output. The code enters a nested loop with two for loops, one for iterating over rows (indexed by *i*) and the other for columns (indexed by *j*). Within the nested loop, it checks the value of $b[i][j]$ to determine which operation to perform on the corresponding elements of *a* and *c* matrices. The possible operations are: (1) If $b[i][j]$ is 0, it multiplies the elements of *a* and *c*. (2) If $b[i][j]$ is 1, it adds the elements of *a* and *c*. (3) If $b[i][j]$ is 2, it subtracts the elements of *c* from *a*. (4) If $b[i][j]$ is 3, it subtracts the elements of *a* from *c*. (5) If $b[i][j]$ is 4, it takes the modulus of the elements of *a* and *c*. (6) If $b[i][j]$ is 5, it performs division on the elements of *a* and *c*. (7) If $b[i][j]$ is 6, it sets the element

of c to the corresponding element of a. (8) If $b[i][j]$ is 7, it sets the element of c to the character "#". (9) If $b[i][j]$ is 8, then the code states the *pass* keyword, which indicates to the reader that in that place any other operation can be inserted for experimentation. (10) If $b[i][j]$ is 9, it checks if the element of c is less than or equal to the element of a and, if so, sets the element of c to the element of a. After each operation, the resulting element in the c matrix is appended to the r string with a space. Also, a newline character is added after each row is processed. At the end, the code prints the content of variable r.

7.1.29 Ex. (123) – Different operations based on maps (SMC)

```
def SMC(m):
    r = "\n"
    for i in range(len(m)):
        for j in range(len(m[i])):
            r += " " + str(m[i][j]) + " "
        r += "\n"
    return r

a = [
    [2, 2, 2, 2, 2],
    [2, 2, 2, 2, 2],
    [2, 2, 2, 2, 2],
    [2, 2, 2, 2, 2],
    [2, 2, 2, 2, 2]
    ]

b = [
    [1, 1, 0, 0, 0],
    [3, 1, 0, 0, 1],
    [1, 3, 0, 1, 1],
    [0, 0, 0, 7, 0],
    [3, 0, 4, 0, 9]
    ]

c = [
    [3, 3, 3, 3, 3],
    [3, 3, 3, 3, 3],
    [3, 3, 3, 3, 3],
    [3, 3, 3, 3, 3],
    [3, 3, 3, 3, 1]
    ]

n = len(a)
m = len(a[0])

for i in range(n):
    for j in range(m):
        if b[i][j] == 0:
            c[i][j] = a[i][j] * c[i][j]
        elif b[i][j] == 1:
```

Output:

```
5 5 6 6 6
1 5 6 6 5
5 1 6 5 5
6 6 6 # 6
1 6 2 6 2
```

```
            c[i][j] = a[i][j] + c[i][j]
        elif b[i][j] == 2:
            c[i][j] = a[i][j] - c[i][j]
        elif b[i][j] == 3:
            c[i][j] = c[i][j] - a[i][j]
        elif b[i][j] == 4:
            c[i][j] = a[i][j] % c[i][j]
        elif b[i][j] == 5:
            c[i][j] = a[i][j] / c[i][j]
        elif b[i][j] == 6:
            c[i][j] = a[i][j]
        elif b[i][j] == 7:
            c[i][j] = '#'
        elif b[i][j] == 8:
            pass  # Placeholder for "do stuff"
        elif b[i][j] == 9:
            if c[i][j] <= a[i][j]:
                c[i][j] = a[i][j]

print(SMC(c))
```

This code example is basically the same as the previous one. It involves operations between the values of two matrices *a* and *c* based on a pattern specified by the *b* matrix. However, the result is stored in the *c* matrix, and the *SMC* function is defined to convert and display the *c* matrix. That is, unlike the previous case when the values from the elements of the *c* matrix were formatted as strings and then accumulated in the *r* variable. To better understand functions please read the next chapter.

7.1.30 Ex. (124) – Nested arrays

```
A = ["a", "b", "c"]
B = ["d", "e", "f"]
C = ["g", "h", "i"]

D = [A, B, C]
E = [B, C, A]
F = [C, B, A]

G = [D, E, F]

print(A[0])
print(D[0])
print(G[0])
```

Output:

```
a
a,b,c
a,b,c,d,e,f,g,h,i
```

This code involves the creation of several arrays and the use of nested arrays. Three arrays *A*, *B*, and *C* are declared, each containing three string values. Then, three arrays *D*, *E*, and *F* are defined. These arrays are comprised of references to the previous arrays *A*, *B*, and *C*. Thus, *D* contains [*A*, *B*, *C*], *E* contains [*B*, *C*, *A*], and *F* contains [*C*, *B*, *A*]. Next, an array *G* is created, which contains references to the arrays *D*, *E*, and *F*. Then,

the code prints elements from these arrays using the *print* function: (1) *print*(*A*[0]) will print the first element of array *A*, which is "a". (2) *print*(*D*[0]) will print the first element of array *D*, which is the reference to array *A*. (3) *print*(*G*[0]) will print the first element of array *G*, which is the reference to array *D*.

Functions

8

Functions are a fundamental concept in computer programming and serve as a building block for organizing and reusing code [1]. At their core, functions are self-contained blocks of code that can perform specific tasks or operations when called [1]. They encapsulate a set of instructions, and one can think of them as named boxes that take some input, process it, and produce an output. Functions are a powerful tool for breaking down complex problems into smaller, manageable pieces, making code more modular, readable, and maintainable. In most computer languages, including Python, Java, VB6, C++, and many others, functions are a vital part of the syntax of the language and are used extensively to structure and control the flow of a program. They allow developers to write code that can be reused in various parts of a program, improving code organization and reducing redundancy. Functions come in various flavors, including simple functions that perform a single task, more complex functions that take multiple parameters, and even functions that return other functions. They play a crucial role in making code more efficient, modular, and easier to understand. The following examples are able to show various functions and their applications in more detail.

8.1 Built-In Functions/Methods

When it comes to Python programming, a robust collection of built-in functions and methods is at the readers disposal, allowing the reader to perform a wide array of tasks efficiently. In this set of examples (Built-in functions/Methods), we will explore the practical application of these pre-defined functionalities to handle various common programming tasks [1]. These functions and methods serve as the building blocks that

P. A. Gagniuc, *Coding Examples from Simple to Complex*, Synthesis Lectures
on Computer Science, https://doi.org/10.1007/978-3-031-53812-4_8

enable developers to manipulate data, work with strings, arrays, and more, ultimately empowering them to create dynamic and feature-rich applications. Let us probe into some illustrative examples to showcase the versatility and power of Python built-in functions and methods.

8.1.1 Ex. (125) – Built-in *sin, exp*

```
import math

a = 3.1415
b = math.sin(a)
c = math.exp(math.sin(a))
print(b)
print(c)
```

```
Output:

0.00009265358966049026
1.000092657882137
```

This example calculates the *sine* of the value approximately equal to *pi* and then calculates the *exponential* value of that *sine*. Several operations are being performed. First, a variable *a* is assigned the value 3.1415, which is an approximation of the mathematical constant π (*pi*). This value is then used in subsequent calculations. Next, a variable *b* is assigned the result of applying the *math.sin*() function to *a*. The *math.sin*() function calculates the sine of an angle in radians, thus, *b* will hold the *sine* of the angle approximately equal to 3.1415. Following that, a variable *c* is assigned the result of applying the *math.exp*() function to the sine of *a*. The *math.exp*() function calculates the exponential value of its argument, and in this case, it is applied to the *sine* of *a*. Therefore, *c* will contain the exponential value of the *sine* of the angle approximately equal to 3.1415. Next, the code shows the values of *b* and *c*.

8.1.2 Ex. (126) – Max between two integer variables

```
maxA = 6
maxB = 10
max_value = max(maxA, maxB)

print(max_value)
```

```
Output:

10
```

This code example sets two variables, *maxA* and *maxB*, assigns them specific values, finds the maximum value between them, and shows the result to the console. Initially, the code begins by declaring two variables, *maxA* and *maxB*, and assigning them with numeric values namely 6 and 10, respectively. Next, the code calculates the maximum value between *maxA* and *maxB* using the *max*() function. The *max*() function is a built-in Python function that takes multiple arguments and returns the largest value among them. In this specific case, it is used to find the maximum value between *maxA* and *maxB*. The

result is then stored in a variable named *max_value*. Next, the content of variable *max_value* is then shown in the console for user inspection.

8.1.3 Ex. (127) – Max between two specific elements of an array

```
a = [6, 7, 1, 9]

max_value = max(a[3], a[1])

print(max_value)
```

```
Output:

9
```

The snippet begins by declaring an array *a* containing four numerical elements: 6, 7, 1, and 9. This array is defined using square brackets and elements separated by commas. Next, the code declares a variable *max_value*. It uses the *max* function to find the maximum value between two elements of the array *a*. Specifically, it compares *a*[3] (which is the fourth element in the array, with an index of 3, holding the value 9) and *a*[1] (the second element in the array, with an index of 1, holding the value 7). The *max* function returns the greater of the two values, in this case, 9. Therefore, this example creates an array and finds the maximum value between two elements of the array, then it shows the result in the console.

8.1.4 Ex. (128) – Max over the values from an array

```
a = [6, 7, 1, 9]
b = [2, 5, 1, 1]
maxA = max(a)
maxB = max(b)
print(maxA)
print(maxB)
```

```
Output:

9
5
```

This code begins by defining two arrays, *a* and *b*, each containing a set of numerical values. Variable *a* holds the values 6, 7, 1, and 9, while *b* contains the values 2, 5, 1, and 1. The subsequent lines of code calculate the maximum value within each of these arrays. The *max*(*a*) function determines the maximum value in array *a*, and this result is stored in the variable *maxA*. Similarly, the *max*(*b*) function finds the maximum value in array *b* and assigns it to the *maxB* variable. Next, the code prints the values of *maxA* and *maxB* to the output.

8.1.5 Ex. (129) – Max over two array variables

```
a = [6, 7, 1, 9]
b = [2, 5, 1, 1]
maxA = max(a)
maxB = max(b)
max_value = max(maxA, maxB)

print(max_value)
```

Output:

```
9
```

In this snippet, two arrays, *a* and *b*, are defined. The array a contains the elements 6, 7, 1, and 9, while the array *b* holds the values 2, 5, 1, and 1. Next, the *max* method is used to find the maximum value within each array. This method is called on the *max* function with the apply method used to pass the array as arguments. Essentially, it is a way to find the maximum value within an array, which is stored in the variables *maxA* and *maxB*. Then, another variable named *max_value* is defined and assigned the result of the *max* function, which takes *maxA* and *maxB* as its arguments. This will give us the maximum value among both arrays.

8.1.6 Ex. (130) – Random integers from 0 to 99 in an array

```
import random

a = []
n = 10

for k in range(n):
    a.append(random.randint(0, 99))

print(a)
```

Output:

```
41,72,71,20,2,
8,40,0,99,38
```

This code generates an array *a* with 10 random integers between 0 and 99, and prints it to the console. First, the code it imports the *random* library. Next, the example initializes an empty array called *a* and a variable *n* with a value of 10. It then enters a *for-loop* that will execute 10 times, starting with *k* as 0 and incrementing it in each iteration until it reaches 9. Within the *for-loop*, the code generates a random number between 0 and 99 (inclusive) using the *randint*() function. The resulting random integer is assigned to the *k*-th element of the array *a*. In other words, it populates the array *a* with 10 random integers. Once the *for-loop* completes, the code shows the content of array *a*.

8.1.7 Ex. (131) – Insert random values in the elements of a matrix.

```
import random

p = []
n = 3
m = 3
r = ''

for i in range(n + 1):
    p.append([])
    for j in range(m + 1):
        p[i].append(random.randint(0, 9))
        r += str(p[i][j]) + ' '
    r += '\n'

print(r)
```

Output:

```
3 1 0 7
9 3 0 2
4 3 4 2
5 7 8 3
```

This above code generates a 3 × 3 matrix of random numbers, storing it in the *p* array, and builds a string representation of the matrix in the variable *r*, where each row is separated by a newline character. The example imports the *random* library. Next, it initializes several variables and uses nested loops to generate a matrix of random numbers. The code begins by declaring a few variables: (1) An empty array *p*, which will be used to store a matrix of random numbers. (2) A variable *n* set to the value 3, representing the number of rows in the matrix. (3) Another variable *m* that is set to 3, indicating the number of columns in the matrix. (4) An empty string *r* that will be used to accumulate the matrix elements as strings, with spaces and line breaks. The code then enters a nested loop structure to fill the *p* array with random numbers and build a string representation of the matrix. It uses two for loops. The outer loop, controlled by the variable *i*, iterates from 0 to *n* (inclusive), creating an array *p*[*i*] for each row. Inside the outer loop, there is an inner loop controlled by the variable *j*, which iterates from 0 to *m* (inclusive). This loop populates each row (*p*[*i*]) with random numbers using *randint* from the *random* library, and stores them in the 2D array *p*. The code appends each of these random numbers to the string *r*, followed by a space (‘.’), effectively building a string representation of the matrix row. Once the inner loop completes for each row, the code appends a newline character (“\n”) to the string *r*, creating a new line in the matrix. Next, the content of variable r is shown in to output.

8.1.8 Ex. (132) – Split string to integers by using a delimiter symbol

```
b = []
a = '2#5#7#1#1#2'
b = a.split('#')

print("b = " + str(b))
```

Output:

```
b = 2,5,7,1,1,2
```

In this snippet, there are two main variables being used. The first variable, *b*, is declared as an empty array. The second variable is *a*, that is declared and assigned a string value, namely "2#5#7#1#1#2", which is a sequence of numbers separated by hash "#" symbols. The crucial operation in this code is "*b = a.split*('#')". Here, the string *a* is split into an array of substrings using the "#" character as the delimiter. The resulting substrings are stored in the *b* array. As a result, variable *b* becomes an array with the following elements: ['2', '5', '7', '1', '1', '2']. Lastly, the source code shows the contents of array *b* to the console for user inspection.

8.1.9 Ex. (133) – Split string to array by using the "|" symbol

```
n = []
m = []

c = 'AAAAA|BBBBB|CCCCC'
n = c.split('|')

print(n[2])
```

Output:

```
CCCCC
```

Here, the source code example shows two arrays *n* and *m*. These arrays are initially empty, which means they do not contain any elements at the start. Next, a string variable *c* is defined and initialized with the value "AAAAA|BBBBB|CCCCC". This string contains three segments separated by the "|" character. The *split* method is then applied to the string *c* using the "|" character as the delimiter. This method splits the string into substrings at each occurrence of the delimiter and stores them in the *n* array. Therefore, after this line of code, the *n* array will contain three elements: "AAAAA", "BBBBB", and "CCCCC". Next, the code outputs the element at index 2 of the *n* array, which corresponds to the string "CCCCC".

8.1.10 Ex. (134) – Cascading built-in functions (*split, join, length*)

```
a = "----##----------##--------"
q = "##"

b = len(a)
c = len(a.replace(q, ""))

if c < b:
    print("a contains q")
```

Output:

```
a contains q
```

Here, we are working with two strings, *a* and *q*. The string *a* is initialized with the value "--##-----##----", while *q* is set to "##". Next, we calculate the length of the string *a* and store it in the variable *b*. In this case, *b* will be 24, which is the total number of characters in the string *a*. Then, a transformation is performed on the string *a*. The

split method is used to break the string into an array based on the substring defined in the variable *q* (in this case, "##"), effectively removing all occurrences of "##". After that, the join method is used to concatenate the array elements back into a string. The resulting string is stored in the variable *c*. Thus, *c* represents the length of the string *a* after removing all instances of "##". Once these calculations are made, we have two variables, namely: *b*, which is the original length of *a*, and *c*, which is the modified length of *a* after removing "##". The code then proceeds to check if *c* is less than *b*. If this condition is met, it means that the length of *a* was reduced by removing "##", indicating that "##" was indeed present in *a*. In this case, it will print the message "a contains q." Therefore, this code is essentially checking whether the string *a* contains the substring "##" and, if so, it outputs a message confirming the presence of "##" within *a*.

8.1.11 Ex. (135) – Built-in *sort* function

```
b = [3, 6, 2, 78, 99, 1, 4]
b.sort()

print(b)
```

```
Output:

1,2,3,4,6,78,99
```

This code snippet begins by declaring an array named *b* and populating it with a series of numerical values enclosed within square brackets. The array consists of the following elements: 3, 6, 2, 78, 99, 1, and 4. Following the array initialization, the code proceeds to invoke the *sort*() method on the array *b*. This method is used to arrange the elements within the array in ascending order, meaning the numbers will be sorted from the lowest value to the highest. Next, the content of *b* is printed.

8.2 Making of Functions

The introduction of the concept of user-defined functions is a fundamental aspect of programming and an essential skill for any developer. In this set of examples, we will explore the creation and use of functions in Python. Note that these native examples are functional in most computer languages once the syntax is adapted. Functions are reusable blocks of code that can be customized to perform specific tasks, making the code more organized, modular, and easier to maintain [1]. The definition of functions allows for logical encapsulation, improving code readability, and facilitating code reuse. Here, we explore a series of practical examples that showcase the power and versatility of user-defined functions.

8.2.1 Ex. (136) – Making of a function

```
def compute(x):
    return x + x / x - x * x

a = 10
b = compute(a)

print(b)
```

Output:

```
-89
```

The code above initializes a with the value 10, performs a series of mathematical operations on a inside the *compute* function, and then it prints the result. The mathematical expression in the *compute* function is stated as:

$$compute(x) = \frac{x + x}{x - x \times x}$$

The above expression in meant to showcase the order of operations in a more graphical manner. Note, however, that the expression could be simplified as:

$$compute(x) = 1 + x - x \times x$$

which is equivalent to:

$$compute(x) = 1 + x - x^2$$

Nevertheless, the above example begins by declaring a variable a and assigning it the value 10. It then proceeds to declare another variable b and assigns it the result of a function call, "*compute*(a)". The *compute* function is defined in the code and takes a single parameter x. Inside the function, a mathematical operation is performed on x. It computes $x + x / x - x * x$, which involves addition, division, subtraction, and multiplication operations. Next, the value of b is printed to the console.

8.2.2 Ex. (137) – Making of a function with more than one parameter

```
def mul(a, b):
    return a * b

print(mul(2, 5))
```

Output:

```
10
```

Essentially, this example computes the product of a and b. It defines a simple function and then calls it. The code begins with a call to the *print* function with the argument *mul*(2, 5). Next, the code defines a function named *mul* using the *def* keyword (*def* -

means definition). This *mul* function takes two parameters *a* and *b*, to showcase an example with more than one parameter. Inside the body of the function, it performs a simple arithmetic operation, namely the multiplication of *a* by *b*, and then returns the result.

8.2.3 Ex. (138) – Gauss summation - sum all from 0 to *n*

```
def gs(n):
    return n * (n + 1) // 2

print(gs(100))
```

```
Output:

5050
```

Here, we have a simple function called *gs* (Gauss summation) which takes a single argument, namely *n*. The purpose of this function is to calculate the sum of the first *n* natural numbers. The formula used to compute this sum is derived from a well-known arithmetic progression formula. The code returns the sum of the natural numbers from 1 to *n* using the formula:

$$gs(n) = \frac{n \times (n-1)}{2}$$

To demonstrate how this function works, there is a call to *gs*(100), which passes the value 100 as an argument to the *gs* function. This function call calculates the sum of the first 100 natural numbers and then prints the result. Thus, if one were to execute this code, it would output the sum of natural numbers from 1 to 100, which is 5050, because *gs*(100) would return 100 * (100 + 1) / 2, which equals 5050.

8.2.4 Ex. (139) – Function calls to other functions

```
def daniela(a, b):
    return a + b

def sebastian(a, b):
    p = daniela(a, b)
    return p

def main_app(x, y):
    cc = sebastian(x, y)
    return cc

d = main_app(66, 100)
print(d)
```

```
Output:

166
```

This source code example creates a series of functions that pass arguments and return values to one another. The *main_app* function is the entry point, and it ultimately calculates the sum of the two initial values, 66 and 100, by passing them through the *sebastian*

and *daniela* functions, and the result is stored in variable *d*. Next, the value stored in variable *d* is printed to the console for user inspection.

8.2.5 Ex. (140) – A simple scanner to find the output distribution

```
def compute(x):
    return x + x / x - x * x

def distribution(start, stop):
    t = ""
    for i in range(start, stop):
        t += str(compute(i)) + "\n"
    return t

a = distribution(3, 21)
print(a)
```

Output:

```
-5
-11
-19
-29
-41
-55
-71
-89
-109
-131
-155
-181
-209
-239
-271
-305
-341
-379
```

The above program calculates a distribution of values within a specified range and stores the result in a string variable. The distribution is obtained by applying the *compute* function to each value within the given range and concatenating the results with newline characters in between. For a short description, this code generates a string output representing a distribution of values. It begins by declaring a variable *a* and assigns it the result of a function call to *distribution*(3, 21). Then, it proceeds to print the value of *a* using a *print* function. The *distribution* function is the heart of the program. It takes two parameters, *start* and *stop*, representing the range of values to consider. Inside the function, a variable *t* is initialized as an empty string. A *for-loop* is used to iterate over a range of values from *start* (inclusive) to *stop* (exclusive). During each iteration, the compute function is called with the current value of *i*, and the result is concatenated to the string *t* with a newline character to separate each value. Next, the resulting string *t* is returned. Note that the *compute* function is a simple mathematical operation that takes a single parameter *x*, and it calculates a value based on the formula: $x + x / x - x * x$ (mentioned earlier in this subchapter).

8.2.6 Ex. (141) – Function chaining - nested function calls

```
def c(x):
    return x + x / x - x * x

a = 3
b = c(c(c(c(a))))
b = -b

print(b)
```

Output:

```
756029
```

In this snippet, we have a series of operations and a function defined. First, a variable *a* is assigned the value 3. Then, a variable *b* is assigned the result of calling the function *c* repeatedly with the argument *a*. The function *c* takes a single argument *x* and performs a series of mathematical operations on it, including addition, division, and multiplication. The final value of *b* is negated, making it negative. Next, the value of *b* is shown to the console. Note that repeated calling of a function with its own result is often referred to as "function composition" or "function chaining." In our code, the function *c* is called multiple times with its own result, which effectively chains the function calls together. This can be a useful technique in some scenarios to apply a series of operations or transformations to a value in a concise and readable manner.

8.2.7 Ex. (142) – Function composition

```
a = [1, 2, 3, 4, 5]
t = 0

def c1(t, a):
    return 5 + c2(t, a)

def c2(t, a):
    return c3(t, a) + 5

def c3(t, a):
    s = 1
    return s + c4(t, a)

def c4(t, a):
    return c5(t, a) + c5(t, a)

def c5(t, a):
    for i in a:
        t += i
    return t

b = c1(t, a)
print(b)
```

Output:

```
41
```

In this code snippet, we have a series of functions and variable assignments that perform calculations based on the input values. We have a constant array *a* containing the elements [1, 2, 3, 4, 5]. A variable *t* is initialized with the value 0. Then, a variable *b* is assigned the result of calling the function *c1* with the arguments *t* and *a*. The function *c1* takes two arguments *t* and *a* and returns the result of calling *c2* with the same arguments and adding 5 to it. The function *c2* takes two arguments *t* and *a* and returns the result of calling *c3* with the same arguments and adding 5 to it. The function *c3* takes two arguments *t* and *a*. Inside this function, a local variable *s* is initialized with the value 1. The function then returns the result of calling *c4* with the same arguments and adding *s* to it. The function *c4* takes two arguments *t* and *a* and returns the result of calling *c5* twice with the same arguments and summing the results. The function *c5* takes two arguments *t* and *a*. It then iterates through the elements of array *a*, adding each element to *t*. Then, it returns the modified value of *t*. The code concludes by printing the value of *b* to the console. The type of function calling demonstrated in the code is often referred to as "function composition" or "function chaining," especially when functions are designed to be composed together in a sequence. In our code, *c1* calls *c2*, which calls *c3*, which calls *c4*, and *c4* calls *c5*. This creates a chain of function calls where each function in the chain relies on the results of the previous one to calculate its own result. This pattern is often used to break down complex operations into smaller, more manageable pieces.

8.2.8 Ex. (143) – Global vaiables and constants

```
a = 3.1415  # constant

b = 11      # global variable

def compute():
    x = b
    return x + x / x - x * x

b = compute()
print(f"{b}\n{A}")
```

Output:

```
-109
3.1415
```

This code snippet demonstrates the use of constants, global variables, a function definition, and function execution. It calculates a result based on the value of the global variable *b* and displays it along with the constant *a*. Two declarations are made: A constant *a* with the value 3.1415 and a global variable *b* with the value 11. The global variable *b* is later updated with the result of calling the *compute* function. The *compute* function takes the current value of *b*, assigns it to a local variable *x*, and then performs a series of mathematical operations on *x*, including addition, division, and multiplication. The result of these operations is then returned and assigned to *b* outside of the *compute* function. Then, the *print* function is used to display the value of *b* followed by a newline character ("\n") and the value of the constant *a*. Note that by convention, constants in Python are written

using uppercase. Here, in the example, the constant name is written in lowercase letters to show that naming constants is really just a convention.

8.2.9 Ex. (144) – Pure and impure functions

```
def pure(x):
    return x + x / x - x * x

def inpure(x):
    global a
    a = 11
    return x + x / x - x * x

a = 10

b = pure(a)
print(str(b) + " & " + str(a))

c = inpure(a)
print(str(c) + " & " + str(a))

d = inpure(a)
print(str(d) + " & " + str(a))
```

Output:

```
-89 & 10
-89 & 11
-109 & 11
```

The code illustrates the difference between "pure" and "impure" functions. "Pure" functions only depend on their input and do not modify any external state, while "impure" functions can modify external variables or have side effects. First, a variable *a* is assigned the value 10. Then, a variable *b* is assigned the result of calling the *pure* function with the argument *a*. The *pure* function takes a single argument *x* and performs a series of mathematical operations on it, including addition, division, and multiplication. After that, the value of *b* is printed along with the original value of *a*. Next, a variable *c* is assigned the result of calling the *inpure* function with the argument *a*. The *inpure* function also takes a single argument *x* but, in this case, it also modifies a global variable *a* by setting it to 11 before performing the same mathematical operations as the *pure* function. The value of *c* is printed along with the modified value of *a*. Next, a variable *d* is assigned the result of calling the *inpure* function with the argument *a* once more. Again, the *inpure* function modifies the global variable *a* by setting it to 11 and performs the mathematical operations. The value of *d* is printed along with the modified value of *a*.

8.2.10 Ex. (145) – Procedures vs Functions

```
a = 16

def f(x):
    return x + x / x - x * x

def p():
    global b
    x = a - 11
    b = x + x / x - x * x

b = f(a)
print(b)

p()
print(b)
```

Output:

```
-239
-19
```

In this code, a distinction between procedures and functions is shown. At the beginning of the code, a variable a is assigned the value 16. Then, a variable b is assigned the result of calling the function f with the argument a, and the value of b is printed to the output. The f function takes a single argument x and performs a series of mathematical operations on it, including addition, division, and multiplication. It returns the result of these operations. Next, there is a procedure named p. Procedures are similar to functions, but they do not return a value explicitly. Inside the p procedure, a new variable x is declared and assigned the result of subtracting 11 from the value of a. Then, b is re-assigned the result of a series of mathematical operations on x, including addition, division, and multiplication. Next, the value of b is printed to the output again, but this time within the p procedure. The key difference between functions and procedures is that functions return values, while procedures do not, and in this code, it is demonstrated how they can be used in different contexts.

8.3 Recursion

Function recursion is a powerful concept in the world of programming, including Python. It allows a function to call itself in a repetitive and self-referential manner, solving complex problems by breaking them down into smaller, more manageable sub-problems. This technique is particularly useful when dealing with tasks that exhibit a recursive structure, such as traversing tree-like data structures, calculating factorials, and implementing various sorting algorithms [1]. Therefore, Python provides a flexible environment for implementing recursive functions. Elegant and/or efficient recursive solutions can be created for various computational problems. Here, the principles of function recursion will be clearly explored by fully understanding its benefits, and see how it works through practical examples.

8.3.1 Ex. (146) – Replacement for repeat loops with recursion

```
# replacement for
# repeat loops.

def for_loop(a, b, r):

    a += 1
    # do stuff from
    r += 5
    # to here

    if a >= b:
        return r
    else:
        return for_loop(a, b, r)

a = for_loop(0, 7, 0)
print(a)
```

```
Output:

35
```

Here, we have a custom recursive function called *for_loop* that may serve as a replacement for traditional repeat loops. The variable *a* is assigned the result of calling the *for_loop* function with the initial parameters 0 for *a*, 7 for *b*, and 0 for *r*. The function is intended to simulate the behavior of a repeat loop. Within the *for_loop* function, variable *a* is incremented by 1 and variable *r* is increased by 5. A comment indicates where the actual processing or "do stuff" part would occur, which is not specified in this code. The function checks if *a* is greater than or equal to *b*. If this condition is met, it returns the value of *r*. Otherwise, it calls itself recursively with the updated values of *a*, *b*, and *r*. This recursive process continues until *a* is greater than or equal to *b*. At the end, the result of the *for_loop* function is stored in the variable *a*, and its value is printed to the console or the output destination.

8.3.2 Ex. (147) – Repeat string *n* times recursively

```
def x(c, s, n):
    s += c
    if len(s) >= n:
        return s
    else:
        return x(c, s, n)

a = x("#", "", 10)
print("Repeat:\n[" + a + "]")
```

```
Output:

Repeat:
[##########]
```

This code effectively demonstrates string repetition through recursion. First, the variable *a* is assigned the result of calling the function *x* with the arguments ("#", "", 10). The function *x* is designed to repeat the character *c* (in this case, "#") *n* times and initially

starts with an empty string *s*. It appends the character to the string *s* in each recursive call until the length of *s* is greater than or equal to *n*. After that, there is a *print* statement that displays the value of a wrapped inside square brackets preceded by "Repeat:\n." The *x* function essentially implements a form of recursion to repeat a character a specified number of times. It appends the character to the string and recursively calls itself until the desired length is reached.

8.3.3 Ex. (148) – Sum from 0 to *n* recursively

```
def sum(n):
    if n <= 1:
        return n
    return n + sum(n - 1)

b = sum(23)
print(f"Sum:[{b}]")
```

```
Output:

Sum:[276]
```

This code calculates the sum of numbers from 0 to 23 using a recursive function and then displays the result as part of a message. It starts by defining a variable *b* and assigning it the result of calling the *sum* function with the argument 23. The sum function calculates the sum of numbers from 0 to *n* using recursion. Inside the *sum* function, there is a base case check. If *n* is less than or equal to 1, it returns *n*, which is the base case of the recursion. Otherwise, it returns *n* added to the result of $sum(n - 1)$, which is the recursive call. After calculating the sum, we use the *print* function to display a message that includes the computed sum. The message is constructed by concatenating the string "Sum:[" with the value of *b*, and then adding a closing "]".

8.3.4 Ex. (149) – Factorial from 0 to *n*

```
# factorial from 0 to n.
def factorial(n):
    if n <= 1:
        return n
    else:
        return factorial(n - 1) * n

c = factorial(10)
print("Factorial:\n[" + str(c) + "]")
```

```
Output:

Factorial:
[3628800]
```

Here, this code snippet calculates the factorial of 10 using a recursive function and then prints the result with an informative message. The code starts by calculating the factorial of the number 10 and assigns the result to the variable *c*. The *print* function shows a message indicating "Factorial" and then displays the value of *c*. The *factorial* function is defined to calculate the factorial of a given number *n*. It uses recursion to perform the

calculation. If n is less than or equal to 1, it returns 1. Otherwise, it recursively calls itself with $n - 1$ and multiplies the result by n to calculate the factorial.

8.3.5 Ex. (150) – Generate a sequence recursively

```
def sequence(n, m, i, t):
    m.append(n)
    i += 1

    if i >= t:
        return m
    else:
        return sequence((n-1) + (n-2), m, i, t)

# Testing the function
# and printing the sequence.
d = sequence(5, [], 0, 5)
print("A sequence:\n[", d, "]")
```

Output:

```
A sequence:
[5,7,11,19,35]
```

This code demonstrates the generation of a sequence of numbers using a recursive function and then displays the result as a string. A variable d is assigned the result of calling the *sequence* function with the arguments 5, an empty array [], 0, and 5. Then, a string is printed to the console, which includes the value of d inside a string representation of an array. The *sequence* function takes four parameters: n, m, i, and t. It appends the value of n to the array m at index i and increments i. If i is greater than or equal to t, the function returns the array m. Otherwise, it makes a recursive call to *sequence*, updating n to be the sum of the previous two values of n and continuing to build the sequence. The result is then printed in the output.

8.3.6 Ex. (151) – Generate fibonacci recursively

```
def fibonacci(n, m, t):
    n += 1
    m.append(m[n - 1] + m[n - 2])

    if n >= t:
        return m
    else:
        return fibonacci(n, m, t)

e = fibonacci(2, [0, 1, 2], 5)
print("Fibonacci:\n["+', '.join(map(str, e))+"]")
```

Output:

```
Fibonacci:
[0,1,2,3,5,8]
```

This code demonstrates the use of a recursive function to generate a *Fibonacci* sequence up to a specified length. First, a variable *e* is assigned the result of calling the *fibonacci* function with the arguments 2, [0, 1, 2], and 5. This function calculates the *Fibonacci* sequence, starting with the initial values [0, 1, 2], and generates the sequence up to the specified length of 5. The *fibonacci* function takes three parameters: *n*, *m*, and *t*. It increments *n* and updates the next value in the *m* array by summing the previous two values in the sequence. It continues to do this recursively until *n* is greater than or equal to *t*, at which point it returns the *m* array containing the *Fibonacci* sequence. Next, the *print* function displays the Fibonacci sequence stored in the *e* variable, preceded by the string "Fibonacci:\n[" and followed by "]".

8.3.7 Ex. (152) – Sum all from array recursively

```
def sum_array(n, q, r):
    r += q[n]

    if n <= 0:
        return r
    else:
        return sum_array(n - 1, q, r)

q = [1, 3, 3, 4, 5, 9]
f = sum_array(len(q) - 1, q, 0)
print(f"Sum array: [{f}]")
```

Output:

```
Sum array:[25]
```

This time, the code demonstrates a recursive function for summing the elements of an array variable. First, an array *q* is defined with some numeric elements. Then, the result of calling the *sum_array* function with three arguments is assigned to a variable *f*: the length of the array *q* minus 1, the array *q* itself, and an initial value of 0. The purpose of the *sum_array* function is to recursively sum the elements of the array. The *sum_array* function takes three arguments: *n* represents the current index, *q* is the array, and *r* is the running sum. It then adds the element at the current index to the running sum. If the index

n is less than or equal to 0, the function returns the running sum *r*. Otherwise, it calls itself recursively with a decremented index and the updated running sum. Next, the code prints a message to the console that includes the sum of the array *q*, which is calculated using the *sum_array* function.

Objects

9

Python is a versatile and widely-used computer language that empowers developers to create dynamic and interactive applications, either on desktops, tablets, phones or on the server side [1, 17]. Python is often referred to as an object-oriented language because it revolves around objects. Objects in Python are data structures that group together related properties and methods. These objects can represent real-world entities, abstract concepts, or any structured data. Objects consist of key-value pairs, where each key (or property) has an associated value [1]. Constructors in Python are essentially blueprints for creating objects [1]. They define the structure and properties of objects that will be created based on that constructor [1]. Functions that serve as constructors are typically named with an initial capital letter. Once an object is created from a constructor, one can add methods to it. Methods are functions attached to an object, and they allow programmers to perform actions related to that object [1]. This interesting combination of constructors and methods enables the creation of reusable and organized code, promoting the principles of encapsulation and code modularity. Another important aspect regarding objects is JSON (JavaScript Object Notation). JSON is a lightweight and text-based data interchange format that plays a crucial role in web development. It is not exclusive to JavaScript but is widely used for data storage, configuration, and data exchange between a server and a client. JSON is a human-readable and easy-to-parse format that closely resembles Python objects. It consists of key-value pairs, arrays, and primitive data types. The JSON simplicity and compatibility with various computer languages have made it the de facto choice for transmitting data over the internet. In Python, JSON can be easily converted to Python objects, and vice versa, using built-in functions, making it an essential tool for web developers when working with external data sources or APIs. This chapter presents some examples related to concepts of Python objects, constructors, methods, and JSON.

P. A. Gagniuc, *Coding Examples from Simple to Complex*, Synthesis Lectures
on Computer Science, https://doi.org/10.1007/978-3-031-53812-4_9

9.1 Constructors and Methods

Constructors and methods are fundamental concepts in Python that play a crucial role in defining and manipulating objects, which are the building blocks of Python programs. Constructors in Python are functions used to create and initialize objects. They serve as blueprints for object creation, allowing one to define the structure and properties of an object [1]. When an object is instantiated by the programmer using a constructor, the programmer essentially creates a new instance with its own set of properties and methods. Constructors are a fundamental part of object-oriented programming in Python. Methods, on the other hand, are functions that are defined within objects. These functions provide a way to encapsulate and perform actions or operations related to the object. Methods are crucial for modeling the behavior of objects in Python applications. They can be used to modify object properties, interact with other objects, or perform specific tasks.

9.1.1 Ex. (153) – Using an object constructor

```
class Obj:
    def __init__(self, a, b, c, d):
        self.ax = list(a)
        self.bx = len(self.ax)
        self.cx = c - b
        self.dx = d * c

o1 = Obj("some", 66, 50, 77)
o2 = Obj("text", 85, 48, 77)

print(o1.ax, "|", o2.ax)
print(o1.bx, "|", o2.bx)
print(o1.cx, "|", o2.cx)
print(o1.dx, "|", o2.dx)

o1.bx = 100

print(o1.bx, "|", o2.bx)
```

Output:

```
s,o,m,e | t,e,x,t
4 | 4
-16 | -37
3850 | 3696
100 | 4
```

The above implementation demonstrates object-oriented programming in Python, with the creation of objects, property assignments, and property value modification. The *Obj* function is a constructor function that takes four parameters *a*, *b*, *c*, and *d*. Within the function, *self.ax* is assigned the result of splitting the string *a* into an array of characters,

whereas *self.bx* is assigned with the length of the array *self.ax*. Also, *self.cx* is assigned the result of subtracting *b* from *c*, and *self.dx* is assigned the result of multiplying *d* by *c*. Once the constructor function is defined, two objects (*o1* and *o2*) are created using the *Obj* constructor with different parameter values. The *print* function is then used to display various properties of these objects with some string concatenation. The properties of *o1* and *o2* are printed side by side for comparison. Then, the value of the *bx* property of *o1* is modified to 100, and its new value is printed alongside the unchanged *bx* value of *o2*.

9.1.2 Ex. (154) – An object with three properties and a method (I)

```
# This example creates an
# object with three properties.
# The cx property is a method.

class Obj:
    def __init__(self):
        self.ax = "this"
        self.bx = "text"

    def cx(self):
        return self.ax + " " + self.bx

# Create an instance of the
# Obj class and call the cx method.

obj = Obj()
print(obj.cx())
```

Output:

```
this text
```

In this code snippet, an object is defined. This object, named *obj*, has three properties. The first property, *ax*, is assigned the string value "this." The second property, *bx*, is assigned the string value "text." The third property, *cx*, is unique as it is assigned a function. This function represents the novelty of this example, and it does not take any parameters except *self*. Inside this function, there is a return statement that concatenates the values of the *ax* and *bx* properties of the *obj* using the this keyword, which refers to the current object. The result is a string that combines the values of *ax* and *bx* separated by a space. Once the object and its properties are fully defined, the code then calls the *cx* method of the *obj* object using the *obj.cx*() syntax. This method call will return the concatenated string of "this text."

9.1.3 Ex. (155) – An object with three properties and a method (II)

```
# This example creates an object
# with three properties. The cx
# property is a method.

class MyObject:
    def __init__(self):
        self.ax = "this"
        self.bx = "text"

    def cx(self, g):
        t = self.ax + g + str(len(self.bx))
        return t

# Create an instance
# of the class.

obj = MyObject()

# Use the functions and
# properties of the object.

print(obj.cx("-"))
print(obj.ax)
print(obj.bx)

# Modify the properties.
obj.ax = "super"
obj.bx = "string"

# Print the modified
# properties and the result
# of the function.

print(obj.ax)
print(obj.bx)
print(obj.cx("+"))
```

Output:

```
this-4
this
text
super
string
super+6
```

Here, an object named *MyObject* is defined with three properties. The first property, *ax*, is initialized with the string value "this." The second property, *bx*, is assigned the string value "text." The third property, *cx*, is a method defined as a function that takes one parameter *g*. Thus, the introduction of this parameter is the novelty here. Inside the *cx* method, there is the declaration of a local variable *t* which is assigned a value by concatenating the *ax* property, the value of the *g* parameter, and the length of the *bx* property. Next, the method returns the value stored in the *t* variable. The code then outputs the result of several states of the object in order to showcase the events: (a) *print*(*obj.cx*("−")); calls the *cx* method of the *obj* object with the parameter "−" and prints the result of the method call. The result is the concatenation of "this," the parameter "−", and the length of the "text" property, resulting in a string that contains "this-4." (b) *print*(*obj.ax*); shows the

value of the *ax* property of the obj object, which is "this." (c) *print*(*obj.bx*); outputs the the *bx* value of the *obj* object, which is "text." (d) *obj.ax* = "super"; updates the value of the *ax* property from "this" to "super." (e) *obj.bx* = "string"; updates the value of the *bx* property from "text" to "string." (f) next we have: *print*(*obj.ax*); that shows an updated value for the *ax* property, which is now "super." (g) *print*(*obj.bx*); this prints the updated value of the *bx* property, which is now "string." (h) *print*(*obj.cx*("+")); calls the *cx* method of the *obj* object with the parameter "+" and prints the result of the method call. The result is the concatenation of "super," the parameter "+," and the length of the "string" property, resulting in a string that contains "super+6."

9.1.4 Ex. (156) – An object with complex methods

```
class Obj:
    def __init__(self):
        self.AV = 0
        self.SD = 0
        self.CV = 0

    def dx(self, a):
        n = len(a)
        b = 0
        e = 0

        for j in a:
            b += j

        x = b / n

        for j in a:
            e += (j - x) ** 2

        s = (e / (n - 1)) ** 0.5
        c = s / x

        self.AV = x
        self.SD = s
        self.CV = c


# Creating an instance of Obj
# and applying the dx function.

obj = Obj()
a = [5, 6, 2, 9, 44, 200]
obj.dx(a)

print('AV:', obj.AV)
print('SD:', obj.SD)
print('CV:', obj.CV)
```

Output:

```
AV: 44.33
SD: 77.83
CV: 1.75
```

This is a more useful representation of an object and its properties. Namely, an object named *obj* is defined. This object has four properties: *AV*, *SD*, *CV*, and *dx*. The *AV*, *SD*, and *CV* properties are initially set to the value 0, while the *dx* property is assigned a function. The *dx* function takes one parameter, *a*, which is assumed to be an array. Within the *dx* function, several operations are performed to calculate statistical values based on the input array *a*. First, the length of the array *a* is stored in the variable *n*. Then, two variables *b* and *e*, are initialized to 0. A *for-loop* is used to iterate over the elements of the array *a*, and in each iteration, the values are accumulated into the variable *b*. After the loop, the mean (average) *x* is calculated by dividing the accumulated sum *b* by *n*. Another *for-loop* is used to calculate the sum of squared differences between each element of the array and the mean *x*. This sum is stored in variable *e*. Subsequently, the standard deviation *s* is computed by taking the square root of the variance, which is calculated by dividing *e* by $(n - 1)$. The coefficient of variation *c* is computed by dividing the standard deviation *s* by the mean *x*. Next, the *AV*, *SD*, and *CV* properties of the *obj* object are updated with the calculated values of *x*, *s*, and *c*, respectively. Outside of the object definition, an array *a* is defined with a list of numerical values. The *dx* method of the *obj* object is called with this array *a* as an argument to perform the statistical calculations. Following the calculation, there are three *print* statements that display the results. These statements use the *print* function to output the mean (*AV*), standard deviation (*SD*), and coefficient of variation (*CV*) to the console. Note that variance: *e* / $(n - 1)$. The standard deviation is the square root of the variance. The expression (*e* / $(n - 1)$) ** 0.5 calculates this square root. The use of "** 0.5" is equivalent to using the *math.sqrt*() function. An experimentation with both approaches yields the same results. Raising a number to the power of 0.5 is equivalent to taking its square root. The square root of a number is the inverse operation of squaring that number. Squaring a number means raising it to the power of 2, thus, the inverse operation is to raise it to the power of ½ or 0.5. One can notice further that $q^{0.5} \times q^{0.5} = q^{0.5+0.5} = q^1 = q$, where q is a number. Thus, $q^{0.5}$ is indeed the square root of q, because when it is multiplied by itself it gives back q.

9.1.5 Ex. (157) – Generate multiple objects with methods

```
# Using an object
# constructor and methods.

import math

class Obj:
    def __init__(self):
        self.AV = 0
        self.SD = 0
        self.CV = 0

    def dx(self, a):
        n = len(a)
        b = sum(a)
        x = b / n

        e = sum([(aj - x) ** 2 for aj in a])
        s = math.sqrt(e / (n - 1))
        c = s / x

        self.AV = x
        self.SD = s
        self.CV = c

a = [5, 6, 2, 9, 44, 200]
b = [7, 4, 6, 8, 6, 4]

box1 = Obj()
box2 = Obj()

box1.dx(a)
box2.dx(b)

print('box 1 - AV:', box1.AV)
print('box 1 - SD:', box1.SD)
print('box 1 - CV:', box1.CV)
print('----------------------')
print('box 2 - AV:', box2.AV)
print('box 2 - SD:', box2.SD)
print('box 2 - CV:', box2.CV)
```

Output:

```
box 1 - AV: 44.333333
box 1 - SD: 77.832298
box 1 - CV: 1.7556157
----------------------
box 2 - AV: 5.8333333
box 2 - SD: 1.6020819
box 2 - CV: 0.2746426
```

This code calculates and displays the mean, standard deviation, and coefficient of variation for the two sets of numbers stored in the *box1* and *box2* objects. An object constructor function named *Obj* is defined. This constructor function takes one parameter, *a*, although it is not used within the function. Inside the constructor function, there are three properties: *AV*, *SD*, and *CV*, all initialized to 0. Additionally, there is a method defined as *dx*, which takes an array *a* as a parameter. The *dx* method performs a series of calculations on the array *a*, including calculating the mean, standard deviation, and coefficient of variation, which are stored in the *AV*, *SD*, and *CV* properties, respectively. Once the *Obj* constructor function is defined, two arrays, *a* and *b*, are further defined with sets of

numbers. Then, two objects, *box1* and *box2*, are created using the new *Obj*() constructor, resulting in two instances of the *Obj* object. The *dx* method is then called on both *box1* and *box2*, passing the *a* and *b* arrays as parameters, respectively. This sets the *AV*, *SD*, and *CV* properties of each object to the calculated values. In the end, the code prints out the values of these properties for both *box1* and *box2* using the *print* function.

9.2 JSON

Constructors and methods are fundamental building blocks in the world of software engineering, serving as the essential tools for creating and manipulating data within software applications. Just as constructors are responsible for initializing objects and defining their initial state, methods enable us to interact with and modify those objects during runtime. Now, for data exchange and storage, JSON (JavaScript Object Notation) provides a perfect illustration of this synergy between constructors and methods [1]. JSON is a lightweight data interchange format that relies on key-value pairs to represent structured data. Constructors, in the context of JSON, serve as blueprints for creating complex data structures, while methods are used to access and manipulate the data contained within these structures. In this dynamic interplay, constructors and methods play a pivotal role in using the power of JSON to manage and transmit data efficiently. Thus, below are a number of examples that explain concepts such as serialization and deserialization via the json library (import *json*).

9.2.1 Ex. (158) – Object to JSON

```
# a Python object...
# ... converted into JSON:

import json

obj = {"v1": 1, "v2": 2, "v3": 3}
txt = json.dumps(obj)

print(txt)

# send JSON:
# print("index.php?obj=" + txt);
```

Output:

```
{"v1":1,"v2":2,"v3":3}
```

Here, we have a simple object manipulation operation where a Python object is converted into JSON format that can potentially be a parameter to a URL (please see the comments). First, an object named *obj* is defined using the curly braces notation. This object has three properties: “v1”, “v2”, and “v3”, each assigned a numeric value. Next, the *json.dumps*(*obj*) method is used to convert the *obj* object into a JSON-formatted

string. This JSON representation will look like {"v1":1,"v2":2,"v3":3}. Then, the *txt* variable is assigned the JSON-formatted string produced by *json.dumps*(*obj*). Afterwards, there is a comment that mentions sending the JSON data. The commented line "*print*("index.php?obj=" + txt)" suggests the construction of a URL string that includes the JSON data as a query parameter. To send the JSON data to a URL, one would typically need to use an HTTP request method. The provided code is incomplete in that regard, as it lacks the actual code for sending the JSON data to a specific URL, thus, it is only a hint to the reader. To sum up this example, this code snippet creates a Python object, converts it into a JSON string, and hints at the intention to send that JSON data to a URL.

9.2.2 Ex. (159) – JSON to Object

```
# txt is text received in JSON format.
# Convert JSON into a Python object:

import json

# JSON text.
txt = '{"v1":1,"v2":2,"v3":3}'

# Parse JSON to create
# a Python object.

obj = json.loads(txt)

print(obj['v1'])
print(obj['v2'])
print(obj['v3'])
```

Output:

```
1
2
3
```

Here, the variable *txt* is assigned with a string containing JSON data, which represents an object with three key-value pairs: "v1" with a value of 1, "v2" with a value of 2, and "v3" with a value of 3. The *json.loads*() method is used to convert the JSON-formatted string stored in the *txt* variable into a Python object. This method parses the JSON data and creates a corresponding Python object, which is then stored in the *obj* variable. Three separate *print* statements are used to log the values of the *v1*, *v2*, and *v3* properties of the *obj* object to the console. Thus, when this code is executed, it will take the JSON string in *txt*, convert it into a Python object named *obj*, and then print the values associated with the properties *v1*, *v2*, and *v3* to the console.

9.2.3 Ex. (160) – Anything to object to string

```
a = ["a", "b", "c"]

b = [
    [0, 1, 0],
    [1, 1, 1],
    [0, 1, 0]
]

c = {"c1": a, "c2": b, "c3": 42}

# Create an object-like dictionary.
obj = {"v1": a, "v2": b, "v3": c}

# Serialize the dictionary
# to a JSON string.

import json
txt = json.dumps(obj)

print(txt)
```

Output:

```
{"v1":["a","b","c"],"v2":[[0,1,0],[1,1,1],[0,1,0]],"v3":
{"c1":["a","b","c"],"c2":[[0,1,0],[1,1,1],[0,1,0]],"c3":42}}
```

Several variables and objects are defined and manipulated in this example. Variable *a* is declared as an array containing three string elements: “a,” “b,” and “c.” Variable *b* is declared as a two-dimensional array (a matrix) containing three arrays. Each inner array represents a row in the matrix and consists of integer values. Variable *c* is defined as an object with three properties: "c1," "c2," and "c3." "c1" is assigned the value of the array *a*, "c2" is assigned the value of the two-dimensional array *b*, and "c3" is assigned the integer value 42. For diversity of data, variable *obj* is defined as an object with three properties: "v1," "v2," and "v3." Thus, "v1" is assigned the value of the array *a*, "v2" is assigned the value of the two-dimensional array *b*, and "v3" is assigned the value of the object *c*. The *json.dumps* function is called to convert the *obj* object into a JSON-formatted string. This string represents the structured data in the *obj* object as a text format. The code essentially defines and structures data in the form of arrays, objects, and matrices, and then converts this data into a JSON string using the *json.dumps* function before logging it to the console.

9.2.4 Ex. (161) – Complex string to object (I) - direct nested access

```
import json

txt = '{"v1":["a","b","c"],"v2":' + \
      '[[0,1,0],[1,1,1],[0,1,0]]' + \
      ',"v3":{"c1":["a","b","c"]' + \
      ',"c2":[[0,1,0],[1,1,1],[0' + \
      ',1,0]],"c3":42}}'

obj = json.loads(txt)

print(obj['v1'][1])
print(obj['v2'][0][1])
print(obj['v3']['c2'])
print(obj['v3']['c2'][1][1])
```

```
Output:

b
1
0,1,0,1,1,1,0,1,0
1
```

A variable *txt* is assigned with a JSON-formatted string. This string represents a JSON object with three key-value pairs. The first key, "v1", has an array value ["a", "b", "c"]. The second key, "v2", contains a nested array [[0, 1, 0], [1], [0, 1, 0]]. The third key, "v3", holds another nested JSON object with three key-value pairs: "c1" with an array value ["a", "b", "c"], "c2" with a nested array value [[0, 1, 0], [1], [0, 1, 0]], and "c3" with a numeric value 42. Following the JSON object creation, the code proceeds to parse this JSON string into a Python object using *json.loads*(). After parsing, it demonstrates the use of the object by printing specific values. It prints the second element of the "v1" array, the value at the first index of the first array within "v2", the entire "c2" object within "v3", and finally, the value at the second index of the second array within "c2". The main purpose of this example is to access the nested values found in the main properties of the object.

9.2.5 Ex. (162) – Complex string to object (II) - nested access by reference

```
txt = '{"v1":["a","b","c"],"v2":' + \
      '[[0,1,0],[1,1,1],[0,1,0]]' + \
      ',"v3":{"c1":["a","b","c"]' + \
      ',"c2":[[0,1,0],[1,1,1],[0' + \
      ',1,0]],"c3":42}}'

import json

obj = json.loads(txt)

a = obj['v1']
b = obj['v2']
c = obj['v3']

print(a)
print(b)
print(c['c1'])
print(c['c2'])
print(c['c3'])
```

Output:

```
a,b,c
0,1,0,1,1,1,0,1,0
a,b,c
0,1,0,1,1,1,0,1,0
42
```

A JSON string named *txt* is defined, which contains three key-value pairs. The first key, "v1", maps to an array of three string elements: "a," "b," and "c." The second key, "v2", maps to a nested array of numeric values. This array has three sub-arrays, each containing numeric values. The third key, "v3," maps to an object with three key-value pairs, namely: (i) "c1" is associated with an array that contains three string elements, similar to the array in "v1." (ii) "c2" is associated with a nested array that is structurally similar to "v2," containing numeric values. (iii) "c3" corresponds to a numeric value, which is specifically 42. After defining the JSON string, the code uses *json.loads* to convert it into an object, which is then stored in the variable *obj*. Subsequently, the code extracts specific components from the parsed object: Variable *a* holds the value of *obj.v1*, which is the first array. Variable *b* holds the value of *obj.v2*, which represents the nested array. Also, *c* holds the value of *obj.v3*, which is an object with sub-properties. Next, the code prints these extracted values for user inspection. The main purpose of this example is to assign parts of one object (i.e. *obj*) into a new object (i.e. *c*) and access the nested values found in its properties (i.e. *c1*, *c2*, and *c3*).

9.2.6 Ex. (163) – Make 1D array from parts of an object

```
import json

txt = '{"v1":["a","b","c"],"v2":' + \
      '[[0,1,0],[1,1,1],[0,1,0]]' + \
      ',"v3":{"c1":["a","b","c"]' + \
      ',"c2":[[0,1,0],[1,1,1],[0' + \
      ',1,0]],"c3":42}}'

obj = json.loads(txt)
a = []

for i in obj['v3']['c1']:
    a.append(i)

print(a)
```

Output:

```
a,b,c
```

Here, a one-dimensional array *a* is made based on values found inside a complex object (*obj*). This code begins by defining a variable *txt*, which is a string containing JSON data. The JSON data within *txt* includes three key-value pairs: "v1", "v2", and "v3": (i) The "v1" key has an array as its value containing the elements "a," "b," and "c." (ii) The "v2" key has a nested array as its value, which represents a 3×3 matrix with binary values. (iii) The "v3" key has an object as its value with three key-value pairs: "c1", "c2", and "c3". (iv) The "c1" key has an array as its value, similar to the "v1" array. (v) The "c2" key has a nested array as its value, representing another 3×3 matrix with binary values. (vi) The "c3" key has a numerical value, which is 42. The code then proceeds to parse the JSON string stored in *txt* into an object and assigns it to the variable *obj*. It initializes an empty array *a* and then uses a *for…in loop* to iterate over the elements of the *c1* array within the *v3* object. For each element, it assigns the value to the corresponding index in array *a*.

9.2.7 Ex. (164) – Make a matrix from parts of an object

```
import json

txt = '{"v1":["a","b","c"],"v2":' + \
      '[[0,1,0],[1,1,1],[0,1,0]]' + \
      ',"v3":{"c1":["a","b","c"]' + \
      ',"c2":[[0,1,0],[1,1,1],[0' + \
      ',1,0]],"c3":42}}'

obj = json.loads(txt)
a = []

for i in range(len(obj['v3']['c2'])):
    a.append([])
    for j in range(len(obj['v3']['c2'][i])):
        a[i].append(obj['v3']['c2'][i][j])

def smc(m):
    r = ''
    for i in range(len(m)):
        for j in range(len(m[i])):
            r += str(m[i][j]) + " "
        r += "\n"
    return r

print(smc(a))
```

Output:

```
0 1 0
1 1 1
0 1 0
```

In this last implementation of this subchapter, a matrix *a* is made based on values found inside a complex object (*obj*). A string *txt* is defined, which contains JSON data, similar to the previous example. The JSON data includes three key-value pairs, namely "v1", "v2", and "v3", and their internal structure is the same as in the previous example (i.e. points i–vi). The code then parses the JSON string stored in *txt* into an object and assigns it to the variable *obj*. Next, an empty array *a* is initialized. Next, there is a nested *for-loop*. The outer loop iterates over the elements of the *c2* array within the *v3* object, and for each element, it initializes a sub-array in array *a*. Then, the inner loop iterates over the elements within the sub-arrays and assigns the corresponding values from the *c2* array to the array *a*. The code defines a function called *SMC*(*m*) that was often used before, which takes a matrix (*m*) as an argument. Inside the function, it constructs a string *r*, which represents a matrix in a human-readable format. It uses nested loops to go through the elements of the matrix, appending each element and adding spaces and newline characters to format it as a matrix. Thus, the array *a* is an argument for the *SMC* function, which in turn prints the matrix to the console.

Moderate Examples

10

A spectrum of coding examples exists, that range from the very basic to the highly advanced. Nestled comfortably in between these extremes are moderate examples. These moderate examples strike a balance between simplicity and complexity, making them valuable teaching tools for both beginners looking to expand their knowledge and experienced developers seeking practical insights. Moderate examples often explore intermediate concepts and techniques, offering real-world relevance without overwhelming programmers with intricate complexities. They bridge the gap between introductory code snippets and complex, production-ready applications, making them an ideal starting point for those eager to deepen their understanding of Python. Thus, building interactive applications, working with data, or optimizing our code for performance, moderate examples provide a stepping stone towards becoming a proficient Python developer.

10.1 Load Arrays from Strings

Loading arrays from strings is a common task in computer programming, often used to convert data in string format into a structured array for further processing. In various computer languages, including Python, JavaScript, and others, this operation plays a crucial role in tasks such as data parsing, data import/export, and matrix manipulation. In the examples that follow, we will explore how to load arrays from strings, demonstrating practical use cases and illustrating the process of converting raw data into a more manageable format. These examples will showcase how to parse strings and transform them into structured arrays, unlocking the potential for a wide range of data processing applications.

P. A. Gagniuc, *Coding Examples from Simple to Complex*, Synthesis Lectures on Computer Science, https://doi.org/10.1007/978-3-031-53812-4_10

10.1.1 Ex. (165) – Strings to 1D arrays (I)

```
def main_app():
    a = "10|13|55|56|1|3|123"
    b = "45|33|55|0|1|22|127"

    aa = a.split("|")
    bb = b.split("|")
    cc = []

    for i in range(len(aa)):
        cc.append(daniela(i, aa, bb))

    print(cc)

def daniela(i, aa, bb):
    return int(aa[i])*int(bb[len(aa)-1-i])

d = main_app()
```

Output:

```
1270,286,55,0,55,99,5535
```

This above program defines a series of functions and variables that perform a specific task. It begins by declaring a variable *d* and initializing it by calling the *main_app* function. The *main_app* function is defined, and it sets up two strings, *a* and *b*, which contain a series of numeric values separated by the pipe character "|". Then, it splits these strings into arrays *aa* and *bb* by using the *split* method. After that, an empty array *cc* is declared to store the results of calculations. The code enters a loop that iterates from *i* equal to 0 up to the length of the *aa* array minus 1. During each iteration, the *daniela* function is called with the current index *i*, the *aa* array, and the *bb* array as arguments. The result of this function call is stored in the *cc* array at the same index *i*. The *daniela* function calculates a value by multiplying the numeric values at the current index *i* of the *aa* array with the corresponding value at the reversed index of the *bb* array (the last value of *bb* corresponds to the first value of *aa*, and so on). It then returns this calculated value. Next, and last, the *print* function is called with the *cc* array as an argument in order to show its content to the user.

10.1.2 Ex. (166) – Strings to 1D arrays (II)

```
def main_app():
    a = "10|13|55|56|1|3|123"
    b = "45|33|55|0|1|22|127"

    aa = list(map(int, a.split("|")))
    bb = list(map(int, b.split("|")))
    cc = []

    for i in range(len(aa)):
        cc.append(sebastian(i, aa, bb))

    return cc

def sebastian(i, aa, bb):
    return aa[i] * bb[len(aa) - 1 - i]

d = main_app()
print(d)
```

Output:

```
1270,286,55,0,55,99,5535
```

Just like before, this code example splits two strings of numbers, performs a series of multiplications on corresponding elements from these arrays, and stores the results in a new array, which is then returned by the *main_app* function and printed as *d*. First, it defines a series of functions to perform a specific task. It starts with a *main_app* function, which is the entry point for the program. The code begins by declaring a variable *d* and assigns the result of calling the *main_app* function to it. The *main_app* function is defined next. Inside the *main_app* function, two strings *a* and *b* are defined. These strings contain a series of numbers separated by the "|" character. The *split* method is used to split these strings into arrays, *aa* and *bb*, respectively. A new array *cc* is also initialized, which will be used to store the results of a computation. The code then enters a *for-loop*, iterating from 0 to the length of the *aa* array minus 1. Inside the loop, the values from the *aa* and *bb* arrays are converted from string to integers via the *sebastian* function. The *sebastian* function is called with three arguments: *i*, *aa*, and *bb*. The result of this function call is stored in the *cc* array at index *i*. Note that the *sebastian* function takes the same three parameters: *i*, *aa*, and *bb*, which indicates that the call is made by reference. The *sebastian* function then retrieves elements from the *aa* and *bb* arrays at specific indices, multiplies them as numbers, and returns the result. Then, the *main_app* function returns the *cc* array, which contains the results of the computations. After calling the *main_app* function, the code prints the value of *d*.

10.1.3 Ex. (167) – A 2D array loaded from string

```
def SMC(matrix):
    result = ""
    for row in matrix:
        for item in row:
            result += item + " "
        result += "\n"
    return result

c = 'AAAAA|BBBBB|CCCCC|DDDDD'
n = c.split('|')

m = [list(row) for row in n]

print(SMC(m))
```

Output:

```
A A A A A
B B B B B
C C C C C
D D D D D
```

Here, this code splits a string into a 2D matrix and prints the matrix, where elements are separated by spaces, and rows are separated by newline characters. The code begins by declaring two empty arrays, *n* and *m*, where *n* is meant to hold a string split into parts, and *m* will represent a matrix. The string *c* is defined as:

$$\text{c} = \text{“AAAAA|BBBBB|CCCCC|DDDDD”}$$

Then *c* is split into parts using the *split*(‘|’) method, with the results stored in the *n* array. The code proceeds with a *for-loop* to iterate through the elements in the array *n*. During each iteration, the elements in *n* are further split into individual characters and stored in array *m*, effectively creating a 2D array or matrix. After setting up the *m* matrix, the code calls a function named *SMC* with *m* as an argument and prints the result. The *SMC* function is defined below. This function iterates through the rows and columns of the *m* matrix, building a string *r* that represents the contents of the matrix with space-separated elements in rows and newline characters separating rows. The resulting string is returned from the *SMC* function, and this final string is printed.

10.1.4 Ex. (168) – Load a matrix from a string by using two delimiters

```
# Initialize empty lists.
n = []
m = []

# Input string.
c = '1,2,4,1,0|3,5,6,7,8|1,2,3,4,5|5,4,3,2,1'

def bahdir(c):

    # Referencing the
    # global variable m.

    global m

    n = c.split('|')

    for i in range(len(n)):
        m.append(n[i].split(','))

    for i in range(len(m)):
        for j in range(len(m[i])):
            m[i][j] = int(m[i][j])
    return m

def smc(m):
    r = "\n"
    for i in range(len(m)):
        for j in range(len(m[i])):
            r += " " + str(m[i][j]) + " "
        r += "\n"
    return r

print(smc(bahdir(c)))
```

Output:

```
 1  2  4  1  0
 3  5  6  7  8
 1  2  3  4  5
 5  4  3  2  1
```

This implementation defines a series of variables and functions for manipulating and printing a matrix represented as a string of numbers separated by commas and pipe symbols. Note that the use of the previous example for splitting numbers found in string format, may create some issues. That is, two or three digit numbers would be counted as separate columns and an error may occur. Here, the goal is to be able to store multi-digit numbers in a string if necessary. The code starts by declaring two empty arrays, *n* and *m*, which will be used to store the matrix data. The string *c* is initialized with a specific set of numbers separated by commas and pipe symbols, which represents a matrix. The numbers are organized in rows and columns, with rows separated by pipe symbols and columns separated by commas. The *bahdir* function is defined, which takes the *c* string as its parameter. This function is responsible for parsing the string and converting it into a two-dimensional array, which represents the matrix. It does this by first splitting the input string *c* into an array of rows using the *split*('|') method. Then, it iterates through each row and further splits each row into an array of numbers by using the *split*(',') method.

Next, it converts the numbers from strings to integers using the *int*() function and stores the resulting two-dimensional array in the *m* variable. The *m* array is returned by the function. Next, the *SMC* function is defined, which takes a two-dimensional array *m* as its parameter (just like in the previous example). Thus, this function is responsible for formatting the matrix and returning it as a string. It initializes a string variable *r* with a newline character to create a new line at the beginning of the output. It then iterates through the rows and columns of array *m* using nested *for-loops*, and for each element in the matrix, it appends the element to the *r* string, surrounded by spaces. After each row is processed, a newline character is added to the *r* string to start a new line. The final formatted matrix is stored in the *r* variable, and it is returned as a string. In the last step, the code calls the *bahdir* function to parse the *c* string and convert it into a two-dimensional array and then passes the result to the *SMC* function to format and print the matrix to the console.

10.1.5 Ex. (169) – A function to correctly display a matrix

```
def ps(a, s):
    t = ""
    b = s - (len(str(a)) % s)
    for i in range(b):
        t += " "
    return t

def SMC(m):
    r = "\n"
    for i in range(len(m)):
        for j in range(len(m[i])):
            r += str(m[i][j]) + ps(m[i][j], 3)
        r += "\n"
    return r

m = [
    [20, 4, 60],
    [39, 5, 60],
    [3, 50, 40]
    ]

print(SMC(m))
```

```
Output:

20 4  60
39 5  60
3  50 40
```

The above example answers the following question: How can an array be elegantly displayed if the values in the array elements contain a different number of digits? Thus, the purpose of this code is to format the matrix *m* and return a string representation of it with each element properly spaced, ensuring a uniform appearance across all columns. The *ps* function aids in achieving this formatting by adding the necessary spaces. But how? At this point it is clear from the last two examples that the *SMC* function is aranging the matrix. However, this arrangement alone, does not take into account the presence of more than two digits in the array elements, which may lead to an erroneous display.

For a proper display, the *ps* function is called from inside *SMC*. Function *ps* takes two arguments: *a* (a number) and *s* (the desired width). It calculates the number of spaces required to make the number *a* fit within the specified width *s*. It does this by finding the difference between *s* and the number of characters in the string representation of *a* and then generating a string consisting of that number of spaces. Thus, the function returns this space-filled string, which is added to *r*, and then *r* is printed to the console.

10.1.6 Ex. (170) – A function to load and display matrices

```
def load(c):
    n = c.split('|')
    m = []

    for i in range(len(n)):
        m.append(n[i].split(','))

    for i in range(len(m)):
        for j in range(len(m[i])):
            m[i][j] = int(m[i][j])

    return m

def SMC(m):
    r = ""
    for i in range(len(m)):
        for j in range(len(m[i])):
            r += str(m[i][j]) + ps(m[i][j], 3)
        r += "\n"
    return r

def ps(a, s):
    t = ""
    b = s - (len(str(a)) % s)
    for i in range(b):
        t += " "
    return t

c1 = '12,2,44,1,0|34,5,6,7,8|' + \
     '1,2,3,4,5|5,4,3,2,1'

c2 = '66,5,45,10,10|37,50,60,17,18|' + \
     '10,25,37,4,5|5,4,3,2,1'

c3 = '66,5,45,10,10|37,50,60,17,18|' + \
     '10,25,37,4,5|5,4,3,2,1'

print(SMC(load(c1)))
print(SMC(load(c2)))
print(SMC(load(c3)))
```

Output:

```
12 2  44 1  0
34 5  6  7  8
1  2  3  4  5
5  4  3  2  1

66 5  45 10 10
37 50 60 17 18
10 25 37 4  5
5  4  3  2  1

66 5  45 10 10
37 50 60 17 18
10 25 37 4  5
5  4  3  2  1
```

This Python code is designed to load and process numerical data represented as comma-separated values in a specific format, and then format the data as a string with

specific spacing between the values, ready for display. In other words, this is a combination between the previous example and the loading of matrices from strings. The code starts by declaring three variables, $c1$, $c2$, and $c3$, each containing a string of comma-separated values arranged in rows separated by pipes (i.e., vertical bars; "|"). These strings are essentially representing numerical matrices, and there are three such matrices. First, the *load* function is defined. This function takes a string as input and splits it into a two-dimensional array where the values are separated by commas and rows are separated by vertical bars. Each element in the array is then converted to a numeric value. The function returns this two-dimensional array. The *SMC* function is also defined. It takes a two-dimensional array as input and processes it to format the data in a specific way. Next, it iterates through each element of the array, converting the values to strings and padding them with spaces to ensure each value is a fixed length of 3 characters. The processed values are then concatenated into a string with each row separated by a newline character. The formatted string is returned. Also, the *ps* function is defined, which is called from inside the *SCM* function. It takes a numeric value and a desired string length (s) as input. It calculates the number of spaces required to pad the value to the specified length and returns a string containing those spaces. After defining these functions, the code proceeds to call the *load* function on each of the $c1$, $c2$, and $c3$ strings. The resulting arrays are then passed to the *SMC* function, and the output is printed to the console for user inspection.

10.1.7 Ex. (171) – Load two matrices from strings and make the addition

```
def load(c):
    n = c.split('|')
    m = []

    for i in n:
        m.append([int(x) for x in i.split(',')])

    return m

def SMC(m):
    r = ""
    for row in m:
        for item in row:
            r += str(item) + ps(item, 3)
        r += "\n"
    return r

def ps(a, s):
    t = ""
    b = s - (len(str(a)) % s)
    for _ in range(b):
        t += " "
    return t

c1 = '12,2,44,1,0|34,5,6,7,8|' + \
```

Output:

```
12 2  44 1  0
34 5  6  7  8
1  2  3  4  5
5  4  3  2  1

66 5  45 10 10
37 50 60 17 18
10 25 37 4  5
5  4  3  2  1

78 7  89 11 10
71 55 66 24 26
11 27 40 8  10
10 8  6  4  2
```

```
      '1,2,3,4,5|5,4,3,2,1'

c2 = '66,5,45,10,10|37,50,60,17,18|' + \
      '10,25,37,4,5|5,4,3,2,1'

m1 = load(c1)
m2 = load(c2)

sm = []

print(SMC(m1))
print(SMC(m2))

for i in range(len(m1)):
    sm.append([])
    for j in range(len(m1[i])):
        sm[i].append(m1[i][j] + m2[i][j])

print(SMC(sm))
```

This code example makes use of the previous examples, namely it loads two matrices from strings, calculates their sum element-wise, and prints the original matrices and their sum as formatted strings. The code begins by declaring two strings, $c1$ and $c2$, which represent two matrices with rows and columns separated by vertical bars ("|"). Then, two empty arrays, $m1$ and $m2$, are declared and populated with the matrices loaded from the respective strings using the *load* function. Next, an empty array *sm* is declared to store the sum of the two matrices. The *print* function is used to print the string representation of the matrices $m1$ and $m2$ after they are loaded using the *SMC* function, which formats the matrices into strings. A nested *for-loop* is used to iterate over the elements of $m1$ and $m2$ and calculate the sum of corresponding elements, storing the result in the *sm* matrix. The *SMC* function is defined to format a matrix as a string. It iterates over the elements of the matrix, converts them to strings, and pads them with spaces to ensure consistent column alignment. As before, the *ps* function is called from inside *SMC* function and it represents a utility function that pads a number with spaces to make it a specific length.

10.2 Some Matrix Operations

As specified before, matrix operations play a fundamental role in various fields of mathematics, science, and computer science. To streamline and modularize the process of performing these operations, it is common practice to store them within functions. These functions serve as reusable building blocks that simplify code, improve readability, and enhance the maintainability of programs that involve matrices. In this context, we will explore the concept of matrix operations stored in functions and their significance in solving complex mathematical and computational problems efficiently. This approach not

only promotes code organization but also facilitates the reuse of these operations in different parts of a program, making it a valuable practice in both algorithm development and software engineering.

10.2.1 Ex. (172) – Function to swap diagonal of matrix

```
# Function to swap
# diagonal of matrix.

def swap_diagonal(a):
    n = len(a)
    for i in range(n):
        t = a[i][i]
        a[i][i] = a[i][n-i-1]
        a[i][n-i-1] = t

def smc(m):
    r = "\n"
    for i in range(len(m)):
        for j in range(len(m[i])):
            r += " " + str(m[i][j]) + " "
        r += "\n"
    return r

a = [
    [3, 1, 2],
    [1, 0, 1],
    [2, 1, 3]
    ]

swap_diagonal(a)
print(smc(a))
```

```
Output:

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

This code swaps the diagonals of a matrix and then prints the modified matrix using the custom *SMC* function. First, the *swap_diagonal* function is defined. It takes a matrix a as an argument. Within this function, it calculates the dimension n of the matrix and then loops through the matrix rows using a *for-loop* with the index variable i. During each iteration, it swaps the element at position (i, i) with the element at position $(i, n - i - 1)$. This effectively swaps the diagonal elements of the matrix. Next, the function called *SMC* is defined. It takes a matrix m as an argument and is responsible for generating a string representation of the matrix. It initializes an empty string r with a newline character. It then uses nested *for-loops* to iterate through the rows and columns of the matrix, building a string representation of the matrix with spaces and newline characters to separate the rows. The code then calls the *swap_diagonal* function to swap the diagonals of the a matrix. After the swap, it prints the matrix using the *SMC* function and the *print* function, similar to the previous code.

10.2.2 Ex. (173) – Function to transpose a matrix

```
# Function to transpose a matrix.

def transpose(a):
    n = len(a)
    m = len(a[0])

    for i in range(n):
        for j in range(i, m):
            t = a[j][i]
            a[j][i] = a[i][j]  # swap.
            a[i][j] = t

def smc(m):
    r = "\n"
    for i in range(len(m)):
        for j in range(len(m[i])):
            r += " " + str(m[i][j]) + " "
        r += "\n"
    return r

a = [
    [1, 2, 3, 4],
    [5, 6, 7, 8],
    [9, 0, 2, 3],
    [4, 5, 6, 7]
    ]

transpose(a)
print(smc(a))
```

Output:

```
 1  5  9  4
 2  6  0  5
 3  7  2  6
 4  8  3  7
```

This version of the code defines a matrix *a*, transposes it using the *transpose* function, and then prints the result of the matrix operation using the *SMC* function. The *SMC* function constructs a string representation of the matrix for display. It starts by defining a 2D array *a*, representing a matrix. The *transpose* function is then called with *a* as its argument, followed by a call to the *SMC* function with *a* as its argument, and the result is printed using a *print* function. The *transpose* function accepts a 2D array *a* and performs the transpose operation on it. It calculates the number of rows *n* and columns *m* in the matrix *a*. It then uses two nested loops to iterate through the matrix and swap elements along the main diagonal, effectively transposing the matrix.

10.2.3 Ex. (174) – Function for rotation of a matrix by 90 degre

```
# left rotation of a matrix
# by 90 degree without using
# any extra space.

def rev_column(a):
    n = len(a)
    m = len(a[0])

    for i in range(n):
        j = 0
        k = m - 1
        while j < k:
            t = a[j][i]
            a[j][i] = a[k][i]
            a[k][i] = t
            j += 1
            k -= 1

def transpose(a):
    n = len(a)
    m = len(a[0])

    for i in range(n):
        for j in range(i, m):
            t = a[j][i]
            a[j][i] = a[i][j]
            a[i][j] = t

def smc(m):
    r = "\n"
    for i in range(len(m)):
        for j in range(len(m[i])):
            r += " " + str(m[i][j]) + " "
        r += "\n"
    return r

a = [
    [1, 1, 1, 1],
    [2, 6, 7, 4],
    [2, 0, 2, 4],
    [2, 3, 3, 3]
    ]

print(smc(a))
transpose(a)
print(smc(a))
rev_column(a)
print(smc(a))
```

```
Output:

 1  1  1  1
 2  6  7  4
 2  0  2  4
 2  3  3  3


 1  2  2  2
 1  6  0  3
 1  7  2  3
 1  4  4  3


 1  4  4  3
 1  7  2  3
 1  6  0  3
 1  2  2  2
```

The overall effect of this code is to rotate the *a* matrix by 90 degrees counterclockwise without using any extra space, and then it reverses the columns of the rotated matrix to complete the 90-degree left rotation. The code defines utility functions *transpose* and

rev_column for this purpose. The matrix *a* is defined as a 4×4 grid of numbers. The *transpose* function takes the *a* matrix and transposes it in place. It swaps elements across the main diagonal of the matrix. This is done by looping through the rows and columns, exchanging *a*[*i*][*j*] with *a*[*j*][*i*] for each element where *i* is the row index and *j* is the column index. This effectively transposes the matrix. After transposing the *a* matrix, the *SMC* function is called to print the matrix, showing the result of the transposition. Next, the *rev_column* function is defined to reverse the columns of the *a* matrix in place. It does this by iterating through the rows and using two pointers, *j* and *k*, to swap elements from the leftmost and rightmost columns within each row. In a final step, after reversing the columns of matrix *a*, the *SMC* function is called again to print the matrix in the output, showing the result of the column reversal.

10.3 Logical Operations

Logical operations play a fundamental role in computer science and computer software in particular, enabling the manipulation and evaluation of data through the use of binary logic. These operations form the building blocks for decision-making, data filtering, and conditional control in various software applications and systems. Common logical operations include AND, OR, NOT, XOR, and more, each serving a distinct purpose in processing and analyzing data. Understanding how to apply these operations is essential for both computer scientists and programmers. One powerful way to comprehend and implement logical operations is through simulation using functions. Functions are modular units of code that encapsulate a specific set of tasks that can be reused. This approach promotes code reusability, readability, and simplifies debugging, making it an essential technique in the world of software development. In these examples, we will explore the fundamentals of logical operations, their importance, and how they can be simulated and applied using functions. Thus, this is an exploration of the concept of truth tables, the role of *Boolean* algebra, and of the examples of how functions can perform logical operations.

10.3.1 Ex. (175) – Logical NOT

```
def f_not(a):
    return 1 - a

# Alternatively, the other
# versions of the function
# can be used, but they are
# commented out here:

# def f_not(a):
#     return (a + 1) % 2

# def f_not(a):
#     if a == 1:
#         a = 0
#     else:
#         a = 1
#     return a

print('1 -> ' + str(f_not(1)))
print('0 -> ' + str(f_not(0)))
```

Output:

```
1 -> 0
0 -> 1
```

The above code demonstrates a simple implementation of a NOT gate function. It begins by calling this function with two different input values, 1 and 0, and then prints the results. The function *f_not* accepts a single argument *a*, which is the input value. It calculates the NOT operation by subtracting the input *a* from 1 and returns the result. This operation effectively inverts the input, turning 1 into 0 and 0 into 1. There are alternative implementations of a NOT gate that have been commented out in the code, but the primary function being used here is the one described above. The alternatives include using the *modulo* operation or conditional statements to achieve the same logical NOT functionality.

10.3.2 Ex. (176) – Logical AND

```
'''

AND
----------------
Input   Output

A   B   Q
---------
0   0   0
0   1   0
1   0   0
1   1   1

'''

def f_and(a, b):
    return a * b

print('[1, 0] -> ' + str(f_and(1, 0)))
```

Output:

```
[1, 0] -> 0
```

The provided implementation from above defines a simple function named *f_and* that implements the logical AND operation. This function takes two input arguments, *a* and *b*, representing binary values (0 or 1). It then computes the logical AND operation between these two input values and returns the result as the output. The logical AND operation returns 1 only when both of its operands are 1; otherwise, it returns 0. This behavior is represented in a truth table that specifies the output (Q) for all possible combinations of inputs (A and B): (i) When A and B are both 0, the output Q is 0. (ii) When A is 0 and B is 1, the output Q is 0. (iii) When A is 1 and B is 0, the output Q is 0. (iv) When both A and B are 1, the output Q is 1. The code then provides an example usage of the *f_and* function with the input [1, 0], and it prints the result as "[1, 0] $\rightarrow$ 0", which corresponds to the logical AND operation of 1 and 0, resulting in 0 (output).

10.3.3 Ex. (177) – Logical OR

```
'''

OR
----------------
Input   Output

A   B   Q
---------
0   0   0
0   1   1
1   0   1
1   1   1

'''

def f_or(a, b):
    return (a + b) - (a * b)

print(f'[1, 0] -> {f_or(1, 0)}')
```

Output:

```
[1, 0] -> 1
```

This code starts with a comment section that contains a truth table illustrating the logical OR operation for two binary inputs A and B. It shows the input combinations (0 and 1 for A and B) and the corresponding output Q of the logical OR operation. The comment provides a clear representation of the expected behavior of the *f_or* function that follows. The code then proceeds to define a function named *f_or*(a, b) which takes two arguments, a and b. Inside the function, it calculates the logical OR operation for the input values a and b using a mathematical expression $(a + b) - (a \times b)$. This expression effectively computes the OR operation and returns the result. Next, the code prints the result of calling the *f_or* function with the input values [1, 0].

10.3.4 Ex. (178) – Logical NAND (NOT AND)

```
'''

NAND
----------------
Input   Output

A   B   Q
---------
0   0   1
0   1   1
1   0   1
1   1   0

'''

def f_nand(a, b):
    return f_not(f_and(a, b))

def f_not(a):
    return 1 - a

def f_and(a, b):
    return a * b

print('[1, 1] -> ' + str(f_nand(1, 1)))
```

Output:

```
[1, 1] -> 0
```

This implementation from above defines a set of functions to implement the NAND logic gate and other related logic gates, specifically NOT and AND gates. The code begins with a comment block that describes the truth table for the NAND gate, showing its inputs A and B and the corresponding output Q. The truth table specifies that the output Q is 1 when A and B are both 0 or when either A or B is 1. Otherwise, when both A and B are 1, the output Q is 0. The code then proceeds to define the following functions: *f_nand*(*a*, *b*), *f_not*(*a*) and *f_and*(*a*, *b*). The *f_nand*(*a*, *b*) function implements the NAND gate and takes two arguments, *a* and *b*. It returns the result of applying the NAND operation on the inputs *a* and *b*. It does this by first calling the *f_and* function to perform the AND operation on *a* and *b*, and then passing the result to the *f_not* function to invert the result, effectively implementing the NAND operation. Described in the previous examples, the *f_not*(*a*) function implements the NOT gate and takes one argument, *a*. It returns the complement of the input *a*. If *a* is 0, it returns 1, and if *a* is 1, it returns 0. Also, the *f_and*(*a*, *b*) function implements the AND gate and takes two arguments, *a* and *b*. It returns the result of applying the AND operation on the inputs *a* and *b*. It multiplies the values of *a* and *b*, and the result is 1 only if both *a* and *b* are 1; otherwise, it is 0. The code concludes by using the defined functions to demonstrate the functionality of the NAND gate by calling *f_nand*(1, 1).

10.3.5 Ex. (179) – Logical NOR (NOT OR)

```
'''

NOR
----------------
Input   Output

A   B   Q
---------
0   0   1
0   1   0
1   0   0
1   1   0

'''

def f_nor(a, b):
    return f_not(f_or(a, b))

def f_not(a):
    return 1 - a

def f_or(a, b):
    return (a + b) - (a * b)

result = f_nor(0, 0)
print(f'[0, 0] -> {result}')
```

```
Output:

[0, 0] -> 1
```

This code defines a set of functions that implement the NOR (NOT OR) logic gate, a basic digital logic gate with two input variables (A and B) and one output variable (Q). The NOR gate returns a true (1) output only when both of its input variables are false (0). The code begins with a comment block, providing a truth table for the NOR gate, which lists the possible input combinations of A and B along with the resulting output Q. The *print* function is used to display the result of the NOR gate for a specific input combination of A and B. For example, [0, 0] $\rightarrow$ 1 indicates that when both A and B are 0, the NOR gate outputs 1. The code defines several functions to implement the NOR gate. The novel function called *f_nor*(a, b) takes two input arguments, a and b, and calculates the NOR operation by first applying the OR operation using the *f_or* function and then negating the result using the *f_not* function. The function *f_not*(a), showcased prior to this example, takes one input argument a and negates it by subtracting it from 1, effectively converting 1 to 0 and 0 to 1. Also, the function *f_or*(a, b) presented before this current example, takes two input arguments, a and b, and calculates the OR operation. It does so by adding a and b and then subtracting their product ($a \times b$). The result is 1 only if at least one of the input values (a or b) is 1; otherwise, it is 0. Thus, this code defines functions to implement the NOR gate, utilizing the concepts of NOT and OR operations to achieve the desired logic. It also provides a specific example of using the *f_nor* function to evaluate

the NOR gate output for the input combination [0, 0], which naturally evaluates to 1 ([0, 0] $\rightarrow$ 1).

10.3.6 Ex. (180) – Logical XOR

```
'''

XOR
----------------
Input   Output

A   B   Q
---------
0   0   0
0   1   1
1   0   1
1   1   0

'''

def f_xor(a, b):
    return (a + b) - 2 * (a * b)
    # return (a - b) * (a - b)
    # return ((a + b) * (a + b)) % 2

print('[0, 0] ->', f_xor(0, 0))
```

Output:

```
[0, 0] -> 0
```

This Python code defines a function that implements the XOR (exclusive OR) logic operation. XOR takes two binary inputs, A and B, and returns 1 if exactly one of them is 1, and 0 if both are the same (0 or 1). The code starts with a comment section that describes the truth table for XOR, showing the input values A and B and their corresponding output Q. It lists all possible combinations and the expected result for each combination. The code then calls the function *f_xor* to demonstrate the XOR operation on the input [0, 0] and prints the result to the console. The *f_xor* function itself implements the XOR logic using simple arithmetic operations. It takes two arguments, a and b, representing the binary inputs. The function calculates the XOR result by adding a and b together ($a + b$), which can be 0, 1, or 2. Then, subtracting twice the product of a and b from the sum. This effectively handles the XOR logic, ensuring that the result is 1 when only one of a or b is 1, and 0 when both are 0 or both are 1. Please notice that the code provides two alternative implementations as comments, which use different mathematical expressions to achieve the same XOR logic. However, the main implementation with addition and subtraction is active.

10.3.7 Ex. (181) – Logical XNOR

```
'''

XNOR
----------------
Input   Output

A   B   Q
---------
0   0   1
0   1   0
1   0   0
1   1   1

'''

def f_xnor(a, b):
    return f_not(f_xor(a, b))

# The commented-out lines represent
# alternative implementations of XNOR.

# def f_xnor_alternatives(a, b):
# return f_not(f_and(f_not(a), b) + \
#        f_and(a, f_not(b)))

# return f_not(f_or(f_not(a), b) + \
#        f_or(a, f_not(b)))

# return f_not(f_or(a, b)) + (a * b)
# return f_not((a + b) - (a * b)) + (a * b)
# return f_not((a + b) - (a * b) + (a * b))
# return f_not((a + b) - 2 * (a * b))

def f_xor(a, b):
    return (a + b) - 2 * (a * b)

def f_not(a):
    return 1 - a

# The OR and AND functions are
# commented out also:

# def f_or(a, b):
#     return (a + b) - (a * b)

# def f_and(a, b):
#     return a * b

print('[0, 0] ->', f_xnor(0, 0))
```

```
Output:

[0, 0] -> 1
```

This code defines functions for XNOR, XOR, and NOT logical operations and provides a simple way to calculate the XNOR of two input values. The code could be extended to include other logical operations by uncommenting and modifying the relevant functions.

First, the implementation defines a set of functions to implement the XNOR (exclusive NOR) logical operation, and it also includes some related functions for other logical operations like XOR, NOT, AND, and OR. It begins with a comment section that provides a table representing the XNOR truth table, with input values A and B, and their corresponding output value Q. The XNOR operation returns 1 (true) when both A and B are the same (either both 0 or both 1), and it returns 0 (false) when A and B are different. The code then prints the result of applying the XNOR operation to the input [0, 0], which outputs 1. Next, there are several functions defined. The *f_xnor*(*a*, *b*) function is the main XNOR function. It takes two arguments, *a* and *b*, representing the input values. It computes the XNOR operation by first calling the *f_xor* function to get the XOR result and then negating it using the *f_not* function to get the final XNOR result. There are some commented-out alternative implementations that I wish the reader to inspect, namely for XNOR using AND, OR, and other logical operations. This demonstrates the myriad of possibilities of achieving the same result, and definitely worth a look. To continue, *f_xor*(*a*, *b*) function calculates the XOR operation between *a* and *b*. XOR returns 1 when the inputs are different and 0 when they are the same. It is implemented by subtracting twice the product of *a* and *b* from the sum of *a* and *b*. Also, the *f_not*(*a*) function implements the NOT operation, which negates the input *a*. It returns 1 if *a* is 0 and returns 0 if *a* is 1. As mentioned above, there are also commented functions for OR and AND, which are not used in the XNOR implementation, but can be valuable teaching examples for different lectures.

10.4 Miscellaneous

Where the lines between disciplines blur, miscellaneous codes are at the forefront, embracing ambiguity, and the ever-expanding possibilities of software engineering. They are not constrained by labels or boundaries, but rather driven by the limitless potential of technology and the boundless horizons of their own capabilities. In short, miscellaneous codes are those implementations that do not fit clearly in any category, these being considered the useful outliers of the field of science (usually). Note, however, that these codes are not necessarily outliers in the scientific realm, but only outliers because of the structure of this book.

10.4.1 Ex. (182) – Logaritm of *b* in base *a*

```
import math

def log(n, v):
    return math.log(v) / math.log(n)

a = 10  # base.
b = 2   # value.

print(log(a, b))
```

```
Output:

0.30102999566398114
```

This code calculates and prints the logarithm of *b* to the base *a* in the console. In this Python code snippet, two variables are defined: *a* and *b*. Variable *a* is assigned the value 10, and it serves as the base, while variable *b* is assigned the value 2, which is the value used in the subsequent mathematical calculation. The code then invokes a function called *log* and passes two arguments to it: *a* and *b*. The *log* function calculates the logarithm of *b* to the base *a* using the Python *math* library (built-in *math.log* function for natural logarithms). It then uses the formula for logarithm conversion by dividing the natural logarithm of *b* by the natural logarithm of *a*. The result of this calculation is then returned by the function and is printed in the output.

10.4.2 Ex. (183) – Smooth signal

```
def smooth(a):
    n = len(a)
    for i in range(1, n - 1):
        a[i] = (a[i-1] + a[i+1]) / 2
    return a

a = [5, 1, 8, 4, 6, 2, 9, 8]

print(smooth(a))
```

```
Output:

5,6.5,5.2,5.6,3.8,6.4,7.2,8
```

This implementation uses an array *a* containing a sequence of numbers: 5, 1, 8, 4, 6, 2, 9, and 8. This array is passed as an argument to a function called *smooth*. The *smooth* function takes the array as its parameter and performs a smoothing operation on it. It first determines the length of the array and stores it in variable *n*. Then, it enters a *for-loop* that iterates over the elements of the array. However, the loop starts at the second element (index 1) and ends at the second-to-last element (index $n - 2$). Inside the loop, each element at index i is updated by taking the average of the elements at indices $i - 1$ and $i + 1$, effectively smoothing out the values. The result of this smoothing operation is stored in the same array *a*, which is then returned by the function. Next, the code calls the *smooth* function and prints the returned array.

10.4.3 Ex. (184) – Greatest common divisor (GCD)

```
# greatest common divisor (GCD).

def gcd(a, b):
    if a == 0:
        return b

    while b != 0:
        if a > b:
            a -= b
        else:
            b -= a
    return a

print(gcd(45, 12))
```

Output:

```
3
```

This calculates the greatest common divisor (GCD) of two numbers and then prints the result. It defines a function called *gcd* which takes two parameters, *a* and *b*. The code first checks if *a* is equal to 0, and if so, it returns the value of *b* as the GCD. If *a* is not zero, it enters a while loop. Inside the loop, it repeatedly subtracts the smaller of the two numbers from the larger one. This process continues until one of the numbers becomes zero. Once that happens, the GCD is found, and the result is returned. Thus, the code calls the *gcd* function with the values 45 and 12 and prints the result.

10.4.4 Ex. (185) – Pseudo random generator

```
def prandom(x):
    a = 11
    m = 25
    c = 17
    r = ""

    for i in range(10):
        x = (a * x + c) % m
        r += str(x) + ","
    return r

# Seed value
x = 3

print(prandom(x))
```

Output:

```
0,17,4,11,13,10,2,14,21,23,
```

In this code, there is a function named *prandom* that generates a sequence of pseudo-random numbers based on a mathematical formula. The function takes an initial value *x* as a parameter, which is used as a seed for the random number generation. The variables *a*, *m*, and *c* are constants used in the formula. The function initializes an empty string *r* to store the generated numbers. It then enters a loop that runs 10 times. In each iteration,

it updates the value of x using the formula:

$$x = (a \times x + c)\%m$$

It then appends the new value of x to the string r, separated by commas. Once all iterations are completed, the function returns the string r, which contains the sequence of generated pseudo-random numbers. The initial value of x and the constants a, m, and c determine the pattern of the generated numbers, making it pseudo-random in nature. This code essentially demonstrates a simple pseudo-random number generator using a linear congruential generator (LCG) algorithm.

10.4.5 Ex. (186) – Double brute force algorithm (DBFA)

```
# Double Brute Force Algorithm (DBFA).

def block_allocation(L):
    a = 1
    b = 1
    t = 5  # min block length.
    m = 8  # max block length.

    while True:
        a = a + 1
        t = L % a
        r = L - t
        v = r % 2
        t += 1
        if not (t > 3 and v == 0):
            break

    while True:
        m = m + 1
        b = r % m
        if not (b == 0 or m > 1000):
            break

    return m

x = block_allocation(133)
print(x)
```

Output:

```
9
```

The code implements a block allocation algorithm that involves two nested brute force loops to determine the value of m based on the input L, and the result is stored in the variable x. The example implements a *Double Brute Force Algorithm* (DBFA) for block allocation [18]. The point of the algorithm is the calculation of text chunks (blocks) of length m that divides a sequence of length L, such that in the last chunk there will be a minimum of t characters. First, this code defines a function called *block_alocation* that takes an input parameter L, and it also demonstrates the usage of this function. The *block_alocation* function begins by declaring several variables: v, r, a, b, t, and m, and initializes

them with specific values. Within the first *while-loop*, it iteratively calculates values for *a*, *t*, *r*, and *v* until the condition ($t > 3$ and $v == 0$) is no longer satisfied. The loop performs a series of mathematical operations on these variables based on the input value *L*. After the first loop exits, the second *while-loop* is initiated. This loop calculates values for *m* and *b* iteratively based on the calculated value of *r* from the previous loop. It continues until the condition ($b == 0$ or $m > 1000$) is no longer met. The final result is the value of *m*, which is returned as the output of the *block_alocation* function. The main part of the code then initializes a variable *x* with the result of calling the *block_alocation* function with the argument 133, and it prints the value of *x* to the console.

10.4.6 Ex. (187) – Alphabet detection

```
# ALPHABET DETECTION.

def alpha(c):
    a = []
    t = list(c)
    k = len(t)

    for i in range(k):
        q = 1
        for j in range(len(a)):
            if t[i] == a[j]:
                q = 0
        if q == 1:
            a.append(t[i])
    return a

print(alpha('uiuhd87wqsaidhsad'))
```

```
Output:

u,i,h,d,8,7,w,q,s,a
```

Alphabet detection is an algorithm that identifies the unique characters from a sequence of text (ex. input: “ABBBABBACABBA”; output: “ABC”). The code defines a function named *alpha* that takes a single argument *c*, which is expected to be a string. Thus, the purpose of this function is to detect unique characters in the input string and return them as an array. Within the function an empty array *a* is initialized. This array will be used to store unique characters from the input string. The input string *c* is split into an array of individual characters and stored in the variable *t*. The length of the array *t* is stored in the variable *k*. Next, there are two nested loops used to identify and store unique characters in the array *a*. The outer loop iterates over the indices of array *t*, from 0 to *k*. Inside the outer loop, a variable *q* is initialized to 1 (flag variable). The inner loop iterates over the indices of the array *a*, from 0 to the current length of *a*. Within the inner loop, the code checks if the current character in *t* at index *i* is equal to any character in the array *a*. If it is, *q* is set to 0, indicating that the character is not unique. If *q* remains 1 after the inner loop, it means that the character is unique, and it is pushed to the array *a*. After both loops have completed, the function returns the array *a*, which contains all the

unique characters from the input string. The *alpha* function is invoked with the argument sequence “uiuhd87wqsaidhsad”, and the result is printed to the console.

10.4.7 Ex. (188) – Alphabet detection on matrices

```
c = [
    [1, 1, 1, 1, 1, 1, 0, 1, 1, 1],
    [1, 0, 1, 0, 1, 1, 1, 0, 1, 1],
    [1, 1, 1, 0, 1, 1, 0, 1, 0, 1],
    [0, 1, 0, 0, 1, 1, 1, 0, 0, 1],
    [1, 1, 1, 0, 1, 1, 1, 0, 1, 0],
    [1, 0, 1, 1, 1, 1, 0, 1, 0, 0],
    [1, 0, 0, 0, 0, 0, 0, 0, 0, 1],
    [1, 0, 1, 1, 1, 1, 0, 1, 1, 1],
    [1, 1, 0, 0, 0, 0, 1, 0, 0, 1]
    ]

def matrix_alphabet(t):
    a = []
    n = len(t)
    m = len(t[0])

    for i in range(n):
        for j in range(m):
            q = 1
            for k in range(len(a) + 1):
                if k < len(a) and t[i][j] == a[k]:
                    q = 0
            if q == 1:
                a.append(t[i][j])
    return a

print(matrix_alphabet(c))
```

Output:

```
1,0
```

The provided code defines a two-dimensional array *c* representing a binary matrix and a function named *matrix_alphabet* that extracts unique elements from this matrix (mainly just like in the previous example but this time for a 2D structure). The implementation calls the function *matrix_alphabet* with the matrix *c* as an argument and prints the result. The *c* array is a 2D array with 9 rows and 10 columns, containing binary values (i.e., 0 or 1) that likely can represent some sort of pattern or alphabet. The *matrix_alphabet* function takes a 2D array *t* as an argument and aims to find the unique elements from it. It initializes an empty array *a* to store these unique elements and calculates the dimensions of the input matrix *t* with *n* representing the number of rows and *m* representing the number of columns. The function then iterates through each element of *t* and checks if it is already in the array *a*. If not, it adds the element to *a*. The function returns the array *a*, which contains the unique elements from the input matrix. The code contains a *print* statement at the end to display the result of calling the *matrix_alphabet* function with the *c* matrix as an argument. Thus, instead of a sequence, the detection of unique characters was made over a matrix.

10.5 Sorting

Sorting is a fundamental concept in computer science and plays a crucial role in a wide range of applications, from data organization to optimization and search algorithms. At its core, sorting involves the arrangement of elements in a specific order, typically in ascending or descending order. This seemingly simple task has far-reaching implications, as efficiently organized data allows for faster search and retrieval, facilitates data analysis, and enhances the overall performance of various algorithms and systems. The need for sorting arises in diverse fields, from databases and information retrieval systems to scientific computing and everyday tasks like organizing files and lists. Sorting is also a key component in numerous computational problems, such as searching for a specific item in a dataset, identifying duplicates, or solving optimization problems. In this subchapter, we will examine different sorting algorithms, their characteristics, and their real-world applications.

10.5.1 Ex. (189) – Low level native sort and eliminate duplicates (I)

```
a = {}
b = [3, 6, 2, 78, 99, 1, 4]

r = 0
n = len(b)

for i in range(n):
    a[b[i]] = b[i]

m = max(a.keys()) + 1

for j in range(m):
    if j in a:
        b[r] = a[j]
        r += 1

print(b)
```

```
Output:

1,2,3,4,6,78,99
```

The example initializes two arrays, *a* and *b*, with *a* initially being an empty array and *b* containing some numerical values. It then defines two variables, *r* and *n*, where *r* is set to 0, and *n* is assigned the length of array *b*. The code proceeds to enter a loop that iterates through the elements of array *b* using a *for-loop*, where *i* serves as the loop counter. Inside the loop, it assigns the value of $b[i]$ to $a[b[i]]$. This effectively creates a new array *a* where the indices correspond to the values of *b*, and the values in *a* are the same as the corresponding values in *b*. After that, the code calculates the length of the array *a* by getting the maximum value among the elements (as the maximum value is equal to the number of elements), and stores it in the variable *m*. A second *for-loop* begins, with *j* as the loop counter, iterating through the indices of array *a*. Inside this loop, it checks

if there is a non-falsy value at index j in array a. If a non-falsy value is found, it assigns that value to $b[r]$ and increments the value of r. Next, the code prints the contents of array b. Note that this native sorting method works well for number sequences of short ranges and is written for this book. To my knowledge it is not published anywhere but here. Note that the time spent by this method to sort the values of an array is $n + m$, where m represents the maximum value over elements of the array.

10.5.2 Ex. (190) – Low level native sort and eliminate duplicates (II)

```
a = {}
b = [3, 6, 2, 78, 99, 1, 4]

n = len(b)
r = n

for i in range(n):
    a[b[i]] = b[i]

m = max(a.keys())

for j in range(m, 0, -1):
    if j in a:
        b[n-r] = a[j]
        r -= 1

print(b)
```

Output:

```
99,78,6,4,3,2,1
```

How about sorting the values from maximum to minimum? Well, we can simply reverse the output order of the previous result. But, let us write something more elegant than a simple inversion. Like before, this example initializes two arrays, a and b. Array b is assigned a set of numeric values. The code also initializes two variables, r and n, with r initially set to 0 and n being the length of array b. The first part of the method is the same as in the previous example. Namely the code then enters a *for-loop* that iterates from 0 to $n - 1$, and in each iteration, it assigns the value of $b[i]$ to the corresponding index in array a. This operation effectively populates array a with values from array b at the same indices. Next, the code calculates the length of array a and stores it in the variable m, representing the maximum value in the array. In the second part, the code then enters another *for-loop*, this time iterating from 0 to $m - 1$. Inside this loop, it checks if the value at index j in array a exists (i.e., is not falsy, or in other words the element is not empty), and if it does, it assigns that value to the corresponding index in array b at index r and increments the value of r. This operation essentially filters out falsy values from array a and stores them in array b. Next, the code prints the modified array b to the console. Note that this native sorting method works well for number sequences containing small maximum values (ex. 100). Again, to my knowledge this method is not published anywhere but here.

10.5.3 Ex. (191) – An optimized version of *Bubble Sort*

```
def bs(a):
    n = len(a)
    for i in range(n - 1):
        for j in range(n - i - 1):
            if a[j] > a[j + 1]:  # swap.
                t = a[j]
                a[j] = a[j + 1]
                a[j + 1] = t
    return a

a = [4, 5, 8, 1, 1, 5, 2, 9]
print(bs(a))

# or a second version:

def bs(a):
    n = len(a)
    for i in range(n-1):
        for j in range(n-i-1):
            if a[j] > a[j+1]:  # swap.
                a[j], a[j+1] = a[j+1], a[j]
    return a

a = [4, 5, 8, 1, 1, 5, 2, 9]
print(bs(a))
```

```
Output:

1,1,2,4,5,5,8,9
```

There are a couple of sorting algorithms. However, this code demonstrates a simple implementation of the *Bubble Sort* algorithm to sort an array a in ascending order. The array a is defined with a set of unsorted numeric values. The code defines a function $bs(a)$ that takes an array a as a parameter and performs the *Bubble Sort* algorithm. Inside the function it initializes variables i, j, n, and t. Variable n is set to the length of the input array a. It uses two nested loops to iterate through the array to compare adjacent elements. If the element at index j is greater than the element at index $j + 1$, it swaps the elements to sort them in ascending order. The t variable is used for temporary storage during the swap. Also, the outer loop (i) iterates $n - 1$ times, and the inner loop (j) iterates $n - i - 1$ times, as the largest elements have already bubbled to the end of the array during each pass of the outer loop. Overall, the sorted array a is returned by the function. Thus, the code calls the $bs(a)$ function with the array a and prints the resulting sorted array to the console for user inspection.

10.6 Permutations

Permutations are a fundamental concept in mathematics and combinatorics. They are at the heart of numerous real-world problems and are essential for understanding the arrangements, orderings, and possibilities that exist within various sets of objects. A permutation refers to an arrangement of objects or elements in a particular order. It represents all the possible ways in which a set of items can be rearranged without repetition. The study of permutations encompasses a wide range of applications, from cryptography and data analysis to game theory, and other related fields.

10.6.1 Ex. (192) – Get all permutations of a given string (I)

```
a = []

def permute(s, r, l):
    if l == r:
        a.append(s)
    else:
        for i in range(l, r + 1):
            s = swap(s, l, i)
            permute(s, r, l + 1)
            s = swap(s, l, i)

def swap(s, i, j):
    c = list(s)
    t = c[i]
    c[i] = c[j]
    c[j] = t
    return ''.join(c)

s = "ACTG"
n = len(s)
permute(s, n - 1, 0)
print(a)
```

```
Output:

ACTG,
ACGT,
ATCG,
ATGC,
AGTC,
AGCT,
CATG,
CAGT,
CTAG,
CTGA,
CGTA,
CGAT,
TCAG,
TCGA,
TACG,
TAGC,
TGAC,
TGCA,
GCTA,
GCAT,
GTCA,
GTAC,
GATC,
GACT
```

This code generates all possible permutations of a given string and stores them in an array *a*. It follows a recursive approach to generate these permutations. The code starts by initializing an empty array *a* to store the permutations. Next, there is a function *permute* that takes three arguments: *s* (the string to be permuted), *r* (the right index), and *l* (the left index). Next, it checks if *l* is equal to *r*, which means that the string is fully permuted. If so, it adds the permuted string *s* to the array *a*. If *l* is not equal to *r*, the function enters a loop that iterates from *l* to *r*. Inside the loop, it swaps characters in the string *s* and recursively calls *permute* with the updated string to permute the remaining characters. After the recursive call, it swaps the characters back to their original positions. There is also a swap function that takes a string *s* and two indices *i* and *j*. It converts the string to an array, swaps the characters at indices *i* and *j*, and then converts the array back to a string. The code defines a string *s* with the initial value "ACTG" and calculates its length *n*. Overall the implementation calls the *permute* function with the string *s*, the right index $n - 1$, and the left index 0 to start the permutation process. Once all permutations are generated the array *a* is shown.

10.6.2 Ex. (193) – Get all permutations of a given string (II)

```
def permute(s, a, b):
    if len(s) == 0:
        b.append(a)
        return

    for i in range(len(s)):
        c = s[i]
        l = s[:i]      # left part.
        r = s[i + 1:]  # right part.
        q = l + r
        permute(q, a + c, b)

s = 'ABC'
a = ''
b = []
permute(s, a, b)

print(b)
```

```
Output:

ABC,
ACB,
BAC,
BCA,
CAB,
CBA
```

Here, there is an array *b* initialized as an empty array. The code defines a function named *permute*, which takes two arguments, *s* and *a*. The purpose of this function is to generate all permutations of a given string *s*. It does this by recursively permuting the characters of the string and collecting the permutations in the array *b*. Within the permute function a few statements allow for the permutations. If the input string *s* is empty, it means a permutation has been successfully formed, and it pushes the current permutation *a* into the *b* array. Otherwise, it iterates over the characters in the string *s*. For each character *c* at index *i*, it splits the string into two parts, *l* (the characters to the left of *c*) and *r* (the characters to the right of *c*). Also, it constructs a new string *q* by combining *l* and *r*, effectively removing *c* from the string. Also, calls the *permute* function recursively with the modified string *q* and the current permutation $a + c$. After defining the *permute* function, the code initializes a string *s* with the value "ABC" and an empty string *a*. Then, it calls the *permute* function with these values, effectively generating all permutations of the string "ABC" and collecting them in the *b* array. Next, it prints the array *b* to the console for the user inspection, which contains all the permutations of the string "ABC".

10.7 Statistics

Statistics is a discipline that lies at the heart of understanding and interpreting data. It is the science of collecting, organizing, analyzing, interpreting, and presenting data to gain insights and make informed decisions [19]. In a world inundated with information, statistics serves as a crucial tool for both scientists and decision-makers, allowing us to extract meaningful patterns and knowledge from the vast array of data that surrounds us. Statistics is not just a collection of mathematical techniques; it is a powerful way of thinking, one that permeates various fields, from science and social sciences to economics and business. Statistics is everywhere, from mathematics to biology [20–22]. Whether it is predicting trends, testing hypotheses in science, or making informed policy decisions, statistics provides the framework for evidence-based reasoning. It helps us answer questions, validate assumptions, and draw meaningful conclusions from raw data, contributing to the advancement of knowledge and informed decision-making [23]. Understanding statistics is not only beneficial for researchers and analysts but also for everyday individuals looking to navigate the increasingly data-driven world effectively.

10.7.1 Ex. (194) – Return an array with proportions (relative frequencies)

```
def p(a):
    max_value = max(a)
    n = len(a)
    m = 100

    # Preallocate the list
    # with the required size.

    t = [''] * n

    for i in range(n):
        t[i] = str(round((m/max_value)*a[i]))+'%'

    return t

a = [5, 1, 8, 4, 6, 2, 9, 8]
print(p(a))


# or another version:


def p(a):
    max_value = max(a)
    n = len(a)
    m = 100
    t = []

    for i in range(n):
        t.append(str(round((m/max_value)*a[i]))+'%')

    return t

a = [5, 1, 8, 4, 6, 2, 9, 8]
print(p(a))
```

```
Output:

56%,11%,89%,44%,
67%,22%,100%,89%
```

This code starts by defining an array named *a*, which contains a list of numeric values. Next, it calls a function named *p* with *a* as an argument and prints the result. The *p* function takes an array *a* as its parameter and performs a number of steps. First, it calculates the maximum value in the input array *a* using *max*(*a*) and stores it in a variable named *max_value*. Next, it determines the length of the input array *a* and stores it in a variable named *n*. Also, it initializes a variable *m* with the value 100. Next, it creates an empty array *t* to store the transformed values. The function enters a loop that iterates over each element in the input array *a*. Inside the loop, it calculates a new value for each element in *t*. The new value is calculated by scaling the original value (*a*[*i*]) by the ratio of *m* to *max_value*. Then it rounds the result to the nearest integer and appends the "%" sign

to it before storing it in the array *t*. The function returns the array *t*, which contains the transformed values of the input array *a*, expressed as percentages. Note that a second version shows how to add elements to array *t* without a precalculated length (i.e., the use of ".append").

10.7.2 Ex. (195) – Average, standard deviation and coeficient of variation

```
def stat(a):
    n = len(a)
    b = 0
    e = 0
    r = [0, 0, 0]  # AV, SD, CV

    for j in range(n):
        b += a[j]

    r[0] = b / n

    for j in range(n):
        e += (a[j] - r[0]) ** 2

    r[1] = (e / (n - 1)) ** 0.5
    r[2] = r[1] / r[0]

    return r

a = [5, 1, 8, 4, 6, 2, 8, 9]
b = stat(a)
print(b)
```

Output:

```
5.375,
2.9246489410818914,
0.5441207332245379
```

The above example performs statistical calculations on an array *a* and then calls the *stat* function with the array. The code starts by defining an array *a* with a list of numerical values. Then, the code initializes a variable *b* to 0 and another variable *e* to 0. Additionally, it creates an array *r* with three elements for storing statistical results, namely for the average (AV; mathematically denoted as $\overline{x}$), standard deviation (SD; mathematically denoted as σ), and the coefficient of variation (CV; mathematically denoted as C_v):

$$\overline{x} = \frac{\sum_{i=1}^{n} x_i}{n} = \frac{\sum_{i=0}^{n-1} a[i]}{n} = \frac{b}{n} = r[0]$$

$$\sigma = \sqrt{\frac{\sum_{i=1}^{n} (x_i - \overline{x})^2}{n-1}} = \sqrt{\frac{\sum_{i=0}^{n-1} (a[i] - r[0])^2}{n-1}} = \sqrt{\frac{e}{n-1}} = \left(\frac{e}{n-1}\right)^{0.5} = r[1]$$

$$C_v = \frac{\sigma}{\overline{x}} = \frac{r[1]}{r[0]} = r[2]$$

For the expressions shown above, please observe the progressive replacement and correlation with the variables from the code. The *for-loop* iterates through each element in the array *a* to calculate the sum of all values, which is stored in variable *b*. Next, the average (AV) is calculated by dividing the sum *b* by the total number of elements in the array (*n*), and the result is stored in *r*[0]. The code then proceeds to calculate the sum of squared differences from the average (*e*) for each element in the array. This step is essential for calculating the standard deviation (SD). The standard deviation (SD) is calculated as the square root of the sum of squared differences from the average, divided by $(n - 1)$. The result is stored in *r*[1]. The coefficient of variation (CV) is calculated as the ratio of the standard deviation to the average (*r*[1]/*r*[0]). The *r* array, which now contains the calculated statistical values, is returned by the *stat* function. Thus, the code prints the result, which is the *r* array returned by the *stat* function, to the console using the *print* function. Note that the *stat* function is essentially used to compute and return statistical information about the input array *a*, including the average, standard deviation, and coefficient of variation. Although clearly described in Ex. 156, here, in order to avoid an unnecessary import for the *math* library and the *sqrt* function, we use pure mathematical knowledge to take the square root. Thus, note that variance: $e/(n - 1)$. The standard deviation is the square root of the variance. The expression $(e/(n - 1))$ ** 0.5 calculates this square root. The use of "** 0.5" is equivalent to using the *math.sqrt*() function of the *math* library. An experimentation with both approaches yields the same results. Raising a number to the power of 0.5 is equivalent to taking its square root. The square root of a number is the inverse operation of squaring that number. Squaring a number means raising it to the power of 2, thus, the inverse operation is to raise it to the power of ½ or 0.5. One can notice further that $q^{0.5} \times q^{0.5} = q^{0.5+0.5} = q^1 = q$, where q is a number. Thus, $q^{0.5}$ is indeed the square root of q, because when it is multiplied by itself it gives back q.

10.7.3 Ex. (196) – Pearson correlation coefficient

```
def p(a, b):
    n = len(a)
    m = [0, 0]

    for i in range(n):
        m[0] += a[i]
        m[1] += b[i]

    m[0] = m[0] / n  # mean a.
    m[1] = m[1] / n  # mean b.

    s0 = 0
    s1 = 0
    s2 = 0

    for i in range(n):
        s0 += (a[i] - m[0]) * (b[i] - m[1])
        s1 += (a[i] - m[0]) ** 2
        s2 += (b[i] - m[1]) ** 2

    r = s0 / (s1 * s2) ** 0.5

    return r

a = [6, 8, 10]
b = [12, 10, 20]

print(p(a, b))
```

Output:

```
0.7559
```

Here, the code begins by initializing two arrays, a and b, each containing three numeric values. The purpose of this code is to calculate the *Pearson correlation coefficient* between these two arrays, which is a statistical measure of the linear relationship between two datasets. The core of the calculation is performed in the p function. Inside this function, the length of the arrays is stored in the variable n. Two arrays, m and $s0$, are initialized to store intermediate values during the calculation. First, the means (averages) of arrays a and b are computed. The sum of all values in a is accumulated in $m[0]$, and the sum of all values in b is accumulated in $m[1]$. These sums are then divided by n to calculate the mean of each array. The next step involves calculating the *Pearson correlation coefficient*, namely r:

$$r = \frac{\sum_{i=1}^{n}[(x_i - \overline{x})(y_i - \overline{y})]}{\sqrt{\sum_{i=1}^{n}(x_i - \overline{x})^2} \times \sqrt{\sum_{i=1}^{n}(y_i - \overline{y})^2}}$$

where x_i is a sample from the first data set, y_i is the sample from the second data set, $\overline{x}$ is the mean of the values from the first data set, $\overline{y}$ is the mean of the values from the second data set, and finally n is the total number of samples from either data set (because they are equal). The formula for r looks complicated, however, the code will

show the reader a different story. Thus, the above mathematical formula is computed by using three accumulators: $s0$, $s1$, and $s2$. Variable $s0$ accumulates the sum of the products of the differences between each element of a and b from their respective means. On the other hand, variable $s1$ accumulates the sum of the squared differences of each element in a from its mean, and $s2$ accumulates the sum of squared differences for each element in b from its mean. With these intermediate values, the *Pearson correlation coefficient* (r) is computed as the ratio of $s0$ divided by the square root of the product of $s1$ and $s2$:

$$r = \frac{s0}{s1 \times s2}$$

$$s0 = \sum_{i=1}^{n}[(x_i - \overline{x})(y_i - \overline{y})] = \sum_{i=0}^{n-1}[(a[i] - m[0])(b[i] - m[1])]$$

$$s1 = \sqrt{\sum_{i=1}^{n}(x_i - \overline{x})^2} = \sqrt{\sum_{i=0}^{n-1}(a[i] - m[0])^2} = \left(\sum_{i=0}^{n-1}(a[i] - m[0])^2\right)^{0.5}$$

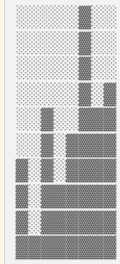
$$s2 = \sqrt{\sum_{i=1}^{n}(y_i - \overline{y})^2} = \sqrt{\sum_{i=0}^{n-1}(b[i] - m[1])^2} = \left(\sum_{i=0}^{n-1}(b[i] - m[1])^2\right)^{0.5}$$

Note the progressive replacement in the formulas with the Python representation from the source code. This coefficient measures the strength and direction of the linear relationship between the two arrays. A positive value of r indicates a positive correlation, a negative r indicates a negative correlation, and r close to 0 indicates a weak or no linear correlation. The result of the *Pearson correlation coefficient* calculation is returned from the p function, and then is printed in the output. In other words, the *Pearson correlation coefficient*, often denoted as "r," it is a common and very important method used to determine how closely two variables are linearly related, with values ranging from -1 (perfect negative correlation) to 1 (perfect positive correlation), and 0 indicating no linear correlation.

10.7.4 Ex. (197) – Vertical chart from the array with pre-declared values

```
def smc(m):
    r = ''
    for row in m:
        for val in row:
            r += val
        r += "\n"
    return r

a = [5, 2, 8, 4, 6, 12, 8, 9]

m = 10
n = len(a)
max_value = max(a)
t = [['' for _ in range(n)] for _ in range(m)]

for j in range(m):
    for i in range(n):
        f = (m * a[i]) // max_value

        # Light shade character.
        t[m-j-1][i] = '\u2591'

        if j < f:
            # Dark shade character.
            t[m-j-1][i] = '\u2593'

print(smc(t))
```

Output:



The application from above shows how an ASCII (American Standard Code for Information Interchange)/UTF-8 chart can be made in the console, with no sophisticated graphical interfaces. The code operates on an array a, which contains a list of numerical values. It also initializes some variables and a 2D array t used for generating a graphical representation of the data. Next, the code calculates the maximum value within array a using the *max* function and stores it in the *max_value* variable. A double *for-loop* is used to populate the 2D array t. The outer loop iterates from 0 to $m - 1$, and the inner loop iterates from 0 to $n - 1$, where m is a scalar variable and n is the length of array a. Inside the inner loop, the code calculates the value f using the formula $(m/max\_value) \times a[i]$, and then it assigns the corresponding character ("\u2591" or "\u2593") to the t array based on the relationship between j and f. The character "\u2591" represents a light shade block, and "\u2593" represents a dark shade block, so this code is essentially generating a bar graph where the darkness of the blocks represents the relative magnitude of the values in the array a. After the construction of the 2D array t, it calls the *SMC* function to convert the array into a string format that visually represents the data in the console.

10.7.5 Ex. (198) – Vertical chart from array with random values at each run

```
import random

a = [0] * 9
n = 9

for k in range(n):
    a[k] = random.randint(0, 99)

def chart(a):
    m = 9
    t = [['' for _ in range(n)] for _ in range(m)]

    # Use default=1 to avoid
    # division by zero.

    max_val = max(a, default=1)

    for j in range(m):
        for i in range(n):

            f = int((m / max_val) * a[i])
            t[m - j - 1][i] = '\u2591'

            if j < f:
                t[m - j - 1][i] = '\u2593'

    return t

def SMC(a):
    n = len(a)
    m = len(a[0])
    r = ''

    for i in range(n + 2):
        for j in range(m):
            if i < n:
                r += a[i][j]
            if i == n:
                r += '|'
            if i > n:
                r += str(j + 1)
        if i == n:
            r += '\n'
        if i < n:
            r += "_" + str(n - i) + "\n"

    return r

print(SMC(chart(a)))
print('\n' + str(a))
```

Output:

```
▒▒▒▒▒▓▒▒▒▒_9
▓▓▒▒▒▓▒▒▒▒_8
▓▓▒▒▓▓▓▒▒▒_7
▓▒▓▓▓▓▓▓▒▒_6
▓▒▓▓▓▓▓▓▓▒_5
▓▒▓▓▓▓▓▓▓▒_4
▓▒▓▓▓▓▓▓▓▓_3
▓▒▓▓▓▓▓▓▓▓_2
▓▓▓▓▓▓▓▓▓▓_1
|||||||||
123456789

79,17,61,61,95,
71,70,56,70,30
```

Compared to the previous example, the new implementation brings the ASCII axis for this chart and the ability to view the chart with different values each time the code is run. Thus, the given code creates a visual representation of data in the form of a bar chart using ASCII characters. The code begins by declaring an empty array *a* and a variable *n* with a value of 9. A *for-loop* iterates from 0 to $n - 1$ (0–8 in this case), generating random integer values between 0 and 99 using *randint*(), and stores them in the array *a*. The source code prints the result of the *SMC*(*chart*(*a*)) function, which generates a bar chart from the array *a* and then prints a visual representation of the chart. The *chart*(*a*) function is defined next. It takes an array *a* as its parameter. Inside the function, it initializes variables *m* and *t*, and it calculates the maximum value in the array *a* by using the *max*(*a*, *default* = 1) function. We use the second parameter as *default* = 1 to avoid division by zero. A nested *for-loop* is able to construct the visual representation of the bar chart, using *Unicode* block characters (▓ ░) to represent the data. The result is stored in the *t* array, which is returned at the end. The *SMC*(*a*) function is defined to create a string representation of the bar chart. It takes an array *a* as its parameter, representing the bar chart. It calculates the dimensions of the chart, initializes an empty string *r*, and then constructs the chart using nested loops. The loops iterate through the array *a* and construct the chart row by row, with underscores ("_") at the bottom, "|" to separate the data from the labels, and labels for each column at the top. The resulting string *r* represents the bar chart and is then returned. In the end, the code prints the chart using *print*(*SMC*(*chart*(*a*))), followed by a new line character and a display of the original data array *a*.

10.7.6 Ex. (199) – Shannon entropy

```
import math

def entropy(c):
    a = alpha(c)
    n = len(a)
    k = len(c)
    e = 0

    for i in range(n):
        r = len(c.replace(a[i], ''))
        a[i] = (k - r) / k
        # e += -a[i] * log(2, a[i])
        e += a[i] * log(2, 1 / a[i])
    return e

# ALPHABET DETECTION.
def alpha(c):
    a = []
    t = list(c)
    k = len(t)

    for i in range(k):
```

Output:

```
3.21
```

```
            q = 1
            for j in range(len(a)):
                if t[i] == a[j]:
                    q = 0
            if q == 1:
                a.append(t[i])
        return a

def log(n, v):
    return math.log(v) / math.log(n)

print(entropy('uiuhd87wqsaidhsad'))
```

The provided code calculates the entropy of a given string and then calls the entropy function with the string "uiuhd87wqsaidhsad". The entropy function is defined to calculate the entropy of a given string c. It starts by calling another function, alpha(c), which is responsible for detecting the alphabet (unique characters) in the input string and returns an array of unique characters. This array is used in subsequent calculations. Inside the *entropy* function, the length of the alphabet is determined by the number of unique characters (stored in variable n), and the total length of the input string is stored in k. The function initializes variables for entropy e, a temporary result string r, and a temporary character l. The main loop then iterates through each unique character in the alphabet (represented by $a[i]$). For each character, it calculates how many times that character appears in the input string c by using a regular expression. The result is stored in r. The character $a[i]$ is then temporarily stored in l. The formula $(k - r)/k$ is used to calculate the probability of that character occurring in the input string. The formula for entropy calculation is then applied, which is essentially the sum of—(p * $\log_2(p)$) for the probability p of each character.

$$e = \sum_{i=1}^{n} p_i \times \log_2\left(\frac{1}{p_i}\right) = \sum_{i=1}^{n} a[i] \times \log_2\left(\frac{1}{a[i]}\right)$$

Or, as it is represented in the commented line from the code:

$$e = -\sum_{i=1}^{n} p_i \times \log_2(p_i) = -\sum_{i=1}^{n} a[i] \times \log_2(a[i])$$

The *alpha* function is responsible for alphabet detection. It initializes an empty array *a* and splits the input string *c* into an array of characters *t*. It then iterates through *t* to find unique characters and appends them to the array *a*. Next, there is a utility function Log(*n*, *v*) that calculates the logarithm of *v* with base *n*. It is used to calculate the entropy. Thus, the code is structured to compute the entropy of the input string, given the alphabet (unique characters) within it. The result is returned by the entropy function, and in this specific case, it is printed to the console using the *print* function.

10.8 Useful Conversions

Conversions between various numerical and text representations are essential in computer science, programming, and digital communication. These conversions enable us to translate data between different formats, making it more accessible, and adaptable for specific applications. Among the most fundamental and commonly used conversions are those between hexadecimal (hex), text, decimal, and binary representations [1]. In this subchapter, we will explore the significance and utility of these conversions, highlighting their relevance in various aspects of computing and data manipulation. Whether it is about encoding characters into binary for digital storage or translating numeric data into a human-readable format, understanding these conversions is crucial for anyone working with digital information.

10.8.1 Ex. (200) – Text (txt) to hexadecimal (hex)

```
# txt to hex.

a = ".~ text"
b = ''
c = [None] * len(a)

for i in range(len(a)):
    b = format(ord(a[i]), 'x')
    c[i] = "0" + b if len(b) < 2 else b

    # if len(b) < 2:
    #     c[i] = "0" + b
    # else:
    #     c[i] = b

print(c)
```

Output:

```
2e,7e,20,74,65,78,74
```

A conversions of text to hexadecimal representation is shown here. The code takes a string *a* containing characters and converts it into a hexadecimal representation. It starts by defining three variables: *a* to store the input string ".~ text," *b* to store temporary values during the conversion, and *c* to store the hexadecimal value of each character. Note that the code is pre-allocating the list with *None* elements. Then, it enters a *for-loop* that iterates through each character in the string *a*. Inside the loop, it uses the *ord* function to get the *Unicode* code point of the current character, and then it converts that code point to a hexadecimal string by using the *format* function (normally the *hex* function would be used instead. However, the point of the exercise is to be as closely as possible to the mirrored twin books in this series). The code checks if the resulting hexadecimal string *b* has a length less than 2. If it does, it adds a leading "0" to ensure that the hexadecimal representation always consists of two characters. The resulting two-character hexadecimal value is then assigned to the corresponding index in the *c* array. Thus, the code prints the *c* array, which contains the hexadecimal representations of the characters from *a*.

10.8.2 Ex. (201) – A *txt* to *hex* from array *a* to array *b*

```
# txt to hex from array a to
# array b element corespondence

a = list(".~ text")
b = [None] * len(a)

for i in range(len(a)):
    b[i] = format(ord(a[i]), 'x')
    b[i] = b[i] if len(b[i])>=2 else "0"+b[i]

print(b)
```

Output:

```
2e,7e,20,74,65,78,74
```

The code from above is a version of the previous example and it involves arrays instead of strings. The implementation is designed to convert characters from an array *a* into their corresponding hexadecimal representations and store them in an array called *b*. The code starts by initializing an array *a* by splitting the string ".~ text" into individual characters and storing them as elements in the array. An empty array *b* is also initialized. A *for-loop* is used to iterate through the elements of array *a*. Inside the loop two events take place. For each character in *a*, the *ord* function is used to retrieve the *Unicode* code point of the character, and the *format* function is then applied to convert it to its hexadecimal representation. The result of the conversion is stored in the corresponding index of array *b*. A conditional statement is used to check if the hexadecimal representation has only one character. If it does, a leading "0" is added to ensure that all representations are two characters long. Next, the code prints array *b* to the console for user inspection, which contains the hexadecimal representations of the characters in the array *a*.

10.8.3 Ex. (202) – A *txt* to *hex* with in-place replacement

```
# txt to hex by replacing each character with
# the hex code in the same element of the array

a = list(".~ text")

for i in range(len(a)):
    a[i] = format(ord(a[i]), 'x')
    a[i] = a[i] if len(a[i])>=2 else "0"+a[i]

print(a)
```

Output:

```
2e,7e,20,74,65,78,74
```

Here, a small optimisation is shown, that is able to use a single array and a conversion in place. As before, the code begins by defining an array *a* which is initialized by splitting the string ".~ text" into individual characters and storing them as separate elements in the array. A *for-loop* is then employed to iterate through the elements of the array *a*. Inside the *for-loop*, the conversion of $a[i]$ is done in place, i.e. replacing the original value in $a[i]$ with the hexadecimal value. Next, the array *a* is printed.

10.8.4 Ex. (203) – A *txt* to *hex* in a function that receives an as argument

```
# txt to hex by in a function
# that recives a as argument

def hexify(a):
    for i in range(len(a)):
        a[i]=format(ord(a[i]), 'x')
        a[i]=a[i] if len(a[i])>=2 else "0"+a[i]
    return a

a = list(".~ text")
print(hexify(a))
```

Output:

```
2e,7e,20,74,65,78,74
```

The current example is the same as the previous one, however, the difference is that a *hexify* function with a parameter *a*, embeds the *for-loop*. In the code outside the function, there is an array *a* initialized with the characters of the string ".~ text" split into individual characters. Then, it calls the *hexify*(*a*) function with this array as an argument and prints the result to the console for user inspection.

10.8.5 Ex. (204) – Multiple functions for any to any conversion

```
def txt_hex(a):
    for i, char in enumerate(a):
        a[i] = format(ord(char), 'x')
        a[i] = a[i] if len(a[i]) >= 2 else "0" + a[i]
    return a

def txt_bin(a):
    for i, char in enumerate(a):
        a[i] = format(ord(char), 'b').zfill(8)
    return a

def txt_dec(a):
    for i, char in enumerate(a):
        a[i] = ord(char)
    return a

def hex_txt(a):
    for i, char in enumerate(a):
        a[i] = chr(int(char, 16))
    return a

def hex_bin(a):
    for i, char in enumerate(a):
        a[i] = bin(int(char, 16))[2:].zfill(8)
    return a
```

```
def hex_dec(a):
    a = hex_txt(a[:])
    for i, char in enumerate(a):
        a[i] = ord(char)
    return a

def bin_hex(a):
    for i, char in enumerate(a):
        a[i] = format(int(char, 2), 'x')
    return a

def bin_txt(a):
    for i, char in enumerate(a):
        a[i] = chr(int(char, 2))
    return a

def bin_dec(a):
    a = bin_txt(a[:])
    for i, char in enumerate(a):
        a[i] = ord(char)
    return a

def dec_hex(a):
    a = dec_txt(a[:])
    return txt_hex(a)

def dec_txt(a):
    for i, char in enumerate(a):
        a[i] = chr(char)
    return a

def dec_bin(a):
    a = dec_txt(a[:])
    return txt_bin(a)

a = list("☁È.~ text")

print('Array a =', a)
print('txt_hex =', txt_hex(a[:]))
print('hex_bin =', hex_bin(txt_hex(a[:])))
print('bin_dec =', bin_dec(txt_bin(a[:])))
print('dec_txt =', dec_txt(txt_dec(a[:])))

print('txt_bin =', txt_bin(a[:]))
print('bin_hex =', bin_hex(txt_bin(a[:])))
print('hex_dec =', hex_dec(txt_hex(a[:])))
print('dec_bin =', dec_bin(txt_dec(a[:])))

print('bin_txt =', bin_txt(txt_bin(a[:])))
print('txt_dec =', txt_dec(a[:]))
print('dec_hex =', dec_hex(txt_dec(a[:])))
print('hex_txt =', hex_txt(txt_hex(a[:])))
```

```
Output:

Array a = ☁,È,.,~, ,t,e,x,t
txt_hex = 2601,400,2e,7e,20,74,65,78,74
hex_bin = 10011000000001,10000000000,101110,1111110,
          100000,1110100,1100101,1111000,1110100
bin_dec = 9729,1024,46,126,32,116,101,120,116
dec_txt = ☁,È,.,~, ,t,e,x,t
txt_bin = 10011000000001,10000000000,101110,1111110,
          100000,1110100,1100101,1111000,1110100
bin_hex = 2601,400,2e,7e,20,74,65,78,74
hex_dec = 9729,1024,46,126,32,116,101,120,116
dec_bin = 10011000000001,10000000000,101110,1111110,
          100000,1110100,1100101,1111000,1110100
bin_txt = ☁,È,.,~, ,t,e,x,t
txt_dec = 9729,1024,46,126,32,116,101,120,116
dec_hex = 2601,400,2e,7e,20,74,65,78,74
hex_txt = ☁,È,.,~, ,t,e,x,t
```

In the previous examples the narrative of the code reached the point of function formation. Here, the code performs a series of character data conversions between different representations, such as hexadecimal, binary, decimal, and plain text. It starts by initializing a string variable *a* with the value "☁È.~ text" and then splits it into an array of individual characters. The code then proceeds to print the original array *a* using the *print* function. Following this, there are a series of conversion functions defined to transform the character data within the array *a*. Each function serves a specific purpose. The *txt_hex*(*a*) function converts each character in the input array *a* into its hexadecimal representation. It does so by utilizing the *ord* function to obtain the character *Unicode* code point and then converting it to hexadecimal format. Next, the *txt_bin*(*a*) function converts each character in the input array *a* into its binary representation. It also employs the *ord* function to obtain the character *Unicode* code point and then converts it to binary format. Next, the *txt_dec*(*a*) function, conversely, converts each character in the input array *a* into its decimal representation. It directly acquires the Unicode code point of each character. Next, the *hex_txt*(*a*) function converts each hexadecimal value in the input array *a* back into its corresponding character. It does so by parsing the hexadecimal string and then using *chr*() to get the character. Next, the *hex_bin*(*a*) function converts each hexadecimal value in the input array *a* into its binary representation. It parses the hexadecimal string and subsequently converts it to binary. The *hex_dec*(*a*) function converts each hexadecimal value in the input array *a* into its decimal representation. It first transforms the hexadecimal string into a character and then retrieves its *Unicode* code point. Next, the *bin_hex*(*a*) function converts each binary value in the input array *a* into its hexadecimal representation. It parses the binary string and then converts it to hexadecimal. Next, the *bin_txt*(*a*) function converts each binary value in the input array *a* into its corresponding

character. It parses the binary string and uses *chr* function to obtain the character. Lastly, the *bin_dec*(*a*) function closes the circle of conversions from anything to anything. It converts each binary value in the input array a into its decimal representation. The function first transforms the binary string into a character and then retrieves its *Unicode* code point. The code finally applies these conversion functions to the array *a*, one by one, and prints the results for each conversion. This code demonstrates how to manipulate character data in various formats, showcasing the flexibility of Python in handling different representations of text characters.

10.8.6 Ex. (205) – One function for any to any conversion and input type detection

```
def convert_to(h, a):
    n = len(a)
    t = sum(len(element) for element in a) / n

    if t == 1:
        q = 'txt'
    elif 2 <= t < 3:
        q = 'hex'
    elif 3 <= t < 4:
        q = 'dec'
    else:
        q = 'bin'

    if q == h:
        return a

    for i in range(len(a)):
        conversion_type = q + '_' + h

        if conversion_type == 'txt_hex':
            a[i] = format(ord(a[i]), 'x').zfill(2)
        elif conversion_type == 'dec_hex':
            a[i] = format(chr(a[i]), 'x').zfill(2)
        elif conversion_type == 'txt_bin':
            a[i] = format(ord(a[i]), 'b')
        elif conversion_type == 'txt_dec':
            a[i] = ord(a[i])
        elif conversion_type == 'hex_dec':
            a[i] = ord(chr(int(a[i], 16)))
        elif conversion_type == 'bin_dec':
            a[i] = ord(chr(int(a[i], 2)))
        elif conversion_type == 'hex_txt':
            a[i] = chr(int(a[i], 16))
        elif conversion_type == 'bin_txt':
            a[i] = chr(int(a[i], 2))
        elif conversion_type == 'dec_txt':
            a[i] = chr(a[i])
        elif conversion_type == 'dec_bin':
```

```
            a[i] = format(ord(chr(a[i])), 'b')
        elif conversion_type == 'hex_bin':
            a[i] = format(int(a[i], 16), 'b')
        elif conversion_type == 'bin_hex':
            a[i] = format(int(a[i], 2), 'x')
    return a

a = list("☁È.~ text")

print('Array a =', a)
print(convert_to('bin', a))
```

```
Output:

Array a = ☁,È,.,~, ,t,e,x,t
10011000000001,10000000000,101110,1111110,100000,1110100,1100101,1
111000,1110100
```

Separate functions for one to one conversions are important pieces of code that can be used as they are in different contexts. However, what about only one function that can convert anything into anything? The following code is another implementation that performs character data conversions between different representations, such as hexadecimal, binary, decimal, and plain text, similar to the previous code. The code starts by initializing a string variable *a* with the value "☁È.~ text" and then splits it into an array of individual characters, just like in the previous version. Then, it uses the *print* function to display the original array *a* for user inspection and comparison. This time, instead of defining multiple conversion functions as in the previous version, this code defines a single function called *convert_to*(*h*, *a*). This function takes two arguments: *h*, which represents the target conversion type (e.g., "bin", "hex", "dec", "txt"), and *a*, which is the input array to be converted. Inside the *convert_to* function, the code calculates the average length of characters in the input array *a* and determines the appropriate conversion type *q* based on this average length. It then checks if the target conversion type *h* is the same as the determined *q*. If they are the same, it returns the input array *a* as there is no need to perform a conversion. If *h* is different from *q*, it iterates through the characters in the array and applies various conversion cases based on the combination of *q* and *h*. These cases cover conversions like text to hexadecimal, text to binary, decimal to text, etc. After performing the necessary conversions, the function returns the modified array *a*. This version of the code is more modular and concise than the previous one. It defines a single conversion function that handles all conversion cases dynamically based on the target and source types. The previous version defined multiple conversion functions, each with a specific purpose, making it longer and potentially harder to maintain. Both versions achieve the same goal of character data conversions, but this version encapsulates the logic within a single function, making it more versatile and adaptable to different conversion scenarios. Additionally, this version calculates the target conversion type dynamically based on the

average character length, which is a feature not present in the previous version. Notice again that the input string type is identified by the first part of the function, which then triggers the conversion case.

10.8.7 Ex. (206) – *Base64* encoding function

```
# Base 64 encoding function.

def encodeBase64(s):
    a = "ABCDEFGHIJKLMNOPQRSTUVWXYZ" + \
        "abcdefghijklmnopqrstuvwxyz0123456789+/"

    b = ""
    for c in s:
        c_bin = format(ord(c), '08b')
        b += c_bin

    # Pad the binary string to make
    # its length a multiple of 6.

    while len(b) % 6 != 0:
        b += "0"

    r = ""
    for i in range(0, len(b), 6):
        x = b[i:i+6]
        d = int(x, 2)
        r += a[d]

    # Add padding if necessary.

    while len(r) % 4 != 0:
        r += "="

    return r

s = "ABC"
q = encodeBase64(s)
print(q)
```

Output:

```
QUJD
```

The code from above defines an encoding function called *encodeBase64* that takes a string *s* as input and returns its Base64 encoding. The code starts by defining a string *s* with the value "ABC." Next, it calls the *encodeBase64* function with *s* as an argument and assigns the result to a variable *q*. The *encodeBase64* function begins by initializing two strings, *a* and *b*. Variable *a* contains the Base64 encoding characters for uppercase letters, lowercase letters, numbers, and two special characters ("+" and "/"), whereas variable *b* is an empty string that will be used to store the binary representation of the characters in the input string. A *for-loop* iterates over each character in the input string *s*. Inside the loop, it converts each character to its binary representation (8 bits) and ensures that the binary representation is left-padded with zeros to make it exactly 8 bits long. The

binary representations are appended to the *b* string. After converting all characters to binary and appending them to *b*, the code checks if the length of *b* is not a multiple of 6. If not, it adds zeros to the end of *b* until its length becomes a multiple of 6. Then, the code initializes an empty string *r*, which will store the final Base64-encoded result. Another *for-loop* iterates over the binary string *b* in chunks of 6 bits at a time. It converts each 6-bit chunk back to decimal and uses the decimal value as an index to look up the corresponding Base64 character from the *a* string. The Base64 characters are appended to the *r* string. Thus, the code checks if the length of the *r* string is not a multiple of 4. If not, it adds "=" padding characters to the end of the *r* string until its length becomes a multiple of 4. The *encodeBase64* function returns the Base64-encoded result *r*. The main part of the code concludes by printing the result *q* to the console, which is the Base64 encoding of the input string "ABC."

Complex Examples

11

Complex examples in Python represent the intricate and multifaceted aspects of this versatile computer language. As Python has evolved over the years, it has become a powerful tool for software development, server-side scripting, and even desktop applications. Whether it is about building interactive web applications, implementing complex algorithms, or integrating with external services, complex Python examples showcase the adaptability of this language and its ability to tackle the demands of modern software development. In this exploration of complexity, developers can uncover innovative solutions and push the boundaries of what is achievable with Python. The following examples demonstrate how Python can be used to create sophisticated and dynamic science and engineering applications and manipulate data, among other advanced functionalities that can be integrated. Note that many of the coding strategies used in the examples of previous chapters will here be fully integrated into different contexts that allow the user to adapt to new cases.

P. A. Gagniuc, *Coding Examples from Simple to Complex*, Synthesis Lectures on Computer Science, https://doi.org/10.1007/978-3-031-53812-4_11

11.1.1 Ex. (207) – Spectral forecast for signals

```
# Spectral forecast for signals in Python.

A = '10.3,23.4,44.8,63.2,44.1,35.1,46.5,62.6,50.4'
B = '18.8,43.1,52.2,45.5,46.8,46.6,67.9,66.3,70.4'
M = ''

tA = A.split(',')
tB = B.split(',')

maxA = max(map(float, tA))
maxB = max(map(float, tB))
max_value = max(maxA, maxB)

d = 33

for i in range(len(tA)):
    v = ((d/maxA)*float(tA[i]))+(((max_value-d)/maxB)*float(tB[i]))
    M += '{:.2f}'.format(v)
    if i < len(tA) - 1:
        M += ','

print('Signal A:' + A)
print('Max(A[i]):' + str(maxA))
print('Signal M:' + M)
print('Signal B:' + B)
print('Max(B[i]):' + str(maxB))
```

Output:

```
Signal A:10.3,23.4,44.8,63.2,44.1,35.1,46.5,62.6,50.4
Max(A[i]):63.2
Signal M:15.37,35.12,51.12,57.17,47.89,43.08,60.35,67.91,63.72
Signal B:18.8,43.1,52.2,45.5,46.8,46.6,67.9,66.3,70.4
Max(B[i]):70.4
```

A spectral forecast implementation for signals is shown above. It processes two input signal arrays, *A* and *B*, and calculates a modified signal *M* based on certain mathematical operations [24]. That is, it creates a signal *M* that resembles signal *A* and *B* in a certain proportion set by variable *d*. The code initially begins by defining three variables: *A*, *B*, and *M*. Variables *A* and *B* are initialized with comma-separated strings of numerical values, representing two input signals. Variable *M* is initialized as an empty string, which will store the modified signal. Two empty arrays, *tA* and *tB*, are declared. The code then uses the *split*() method to split the strings *A* and *B* into arrays *tA* and *tB*, respectively, using commas as the delimiter. Next, it calculates the maximum values in the *tA* and *tB* arrays using the *max*() function. The maximum values are stored in *maxA* and *maxB* variables. The overall maximum value between *maxA* and *maxB* is calculated and stored in the *max_value* variable. Next, a variable *d* is initialized with the value 33. A *for-loop*

is used to iterate through the elements of the *tA* array. Inside the loop, a variable *v* is calculated using a formula that involves scaling the elements of *tA* and *tB* by certain factors based on *maxA*, *maxB*, and *d*. The calculated value *v* is then appended to the *M* string with two decimal places using the '{:0.2f}'.*format*(*v*) method. A comma is added to separate values in the *M* string only if it is not the last element. Post processing, the code prints the following information in the output: (i) The original signal *A*. (ii) The maximum value found in *A*. (iii) The modified signal *M*. (iv) The original signal *B*. (v) The maximum value found in *B*.

11.1.2 Ex. (208) – Logic gate functions applied to matrix elements

```
def f_not(a):
    return 1 - a

def f_and(a, b):
    return a * b

def f_or(a, b):
    return (a + b) - (a * b)

def f_nand(a, b):
    return f_not(f_and(a, b))

def f_nor(a, b):
    return f_not(f_or(a, b))

def f_xor(a, b):
    return (a + b) - 2 * (a * b)

def f_xnor(a, b):
    return f_not(f_xor(a, b))

a = [
    [1, 1, 1],
    [0, 1, 0],
    [0, 1, 0]
    ]

b = [
    [0, 1, 0],
    [1, 1, 1],
    [0, 1, 0]
    ]
```

Output:

```
0 1 0
0 1 0
1 1 1
```

```
c = [
    [0, 0, 0],
    [0, 0, 0],
    [0, 0, 0]
    ]

n = len(a)
m = len(a[0])
r = ''

for i in range(n):
    r += '\n'
    for j in range(m):
        c[i][j] = f_xnor(a[i][j], b[i][j])
        r += str(c[i][j]) + " "

print(r)
```

This example defines and performs operations on matrices *a*, *b*, and *c* using various logical functions. The code begins by initializing three matrices *a*, *b*, and *c*, where each matrix is represented as an array of arrays, containing numeric values. Variables *n* and *m* are set to the number of rows and columns in matrix *a*, respectively. An empty string *r* is initialized, which will be used to build a string representation of the result. Next, a nested *for-loop* is used to iterate over each element in matrices *a* and *b*. Inside the loop, the *f_xnor* function is called with the corresponding elements from *a* and *b*. The result of the *f_xnor* function is stored in the corresponding position of matrix *c*, and the result is also appended to the string *r* with a space. The *f_xnor* function is defined to calculate the XNOR (exclusive NOR) operation between two values. It calls the *f_xor* function and then negates the result using the *f_not* function. Below the main code logic, several logical functions are defined. Function *f_not*(*a*) that returns the logical NOT operation of a value. Next, function *f_and*(*a*, *b*) is defined, that returns the logical AND operation between two values. Also, function *f_or*(*a*, *b*) is defined, that is able to return the logical OR operation between two values. Next, function *f_nand*(*a*, *b*) returns the logical NAND operation between two values by combining the *f_and* and *f_not* functions. Next the *f_nor*(*a*, *b*) function returns the logical NOR operation between two values by combining the *f_or* and *f_not* functions. Also, function *f_xor*(*a*, *b*) is defined, that returns the logical XOR operation between two values. Lastly, the *f_xnor*(*a*, *b*) function is defined, that is able to return the logical XNOR operation between two values by combining the *f_xor* and *f_not* functions.

11.1.3 Ex. (209) – The general logic gate based on a map

```
def g(a, b, t):
    h = [
        [0, 0, 0, 1, 0, 1, 0, 1],
        [0, 1, 0, 1, 1, 0, 1, 0],
        [1, 0, 0, 1, 1, 0, 1, 0],
        [1, 1, 1, 0, 1, 0, 0, 1]
        ]

    for i in range(len(h)):
        if a == h[i][0] and b == h[i][1]:
            return h[i][t + 1]

a = [
    [1, 1, 1],
    [0, 1, 0],
    [0, 1, 0]
    ]

b = [
    [0, 1, 0],
    [1, 1, 1],
    [0, 1, 0]
    ]

c = []
n = len(a)
m = len(a[0])
r = ''

for i in range(n):
    r += '\n'
    c.append([])
    for j in range(m):
        c[i].append(g(a[i][j], b[i][j], 6))
        r += str(c[i][j]) + " "

print(r)

'''
_____________________________
|       |   | N |   |   | X |
| Input | A | A |   | N | X | N |

| a | b | D | D | R | R | R | R |
-------------------------------
| 0 | 0 | 0 | 1 | 0 | 1 | 0 | 1 |
| 0 | 1 | 0 | 1 | 1 | 0 | 1 | 0 |
| 1 | 0 | 0 | 1 | 1 | 0 | 1 | 0 |
| 1 | 1 | 1 | 0 | 1 | 0 | 0 | 1 |
-------------------------------
'''
```

Output:

```
0 1 0
0 1 0
1 1 1
```

The previous example showed independent scattered functions for the logic gate functions. However, how about an optimization that makes a shortcut that melts all the previously described functions into one? Well, this code defines two 3×3 matrices a and

b as well as an empty matrix c. It then performs bitwise logical operations between the corresponding elements of matrices a and b based on a specified operation code t using a function called g. The result of these operations is stored in matrix c. Next, it prints the contents of matrix c in a human-readable format. In detail, the matrices a and b are 3×3 arrays of binary values. The code initializes an empty matrix c, and two variables n and m to store the dimensions of matrix a. It also initializes an empty string r to store the formatted result. A nested *for-loop* is used to iterate through the elements of matrices a and b, perform the logical operation using the g function, and store the result in the matrix c. It also constructs a string r that represents the elements of matrix c separated by spaces and newlines. The g function takes three arguments: a, b, and t. It uses a predefined matrix h to perform bitwise logical operations based on the value of t and returns the result. The matrix h is a lookup table that specifies the results of different logical operations (AND, NAND, OR, NOR, XOR, XNOR) for different combinations of a and b. The code concludes by printing the string r, which represents the resulting matrix c. Additionally, there are comments in the code that explain the meanings of the values of t and provide a visual representation of the logical operations and their results in a tabular format.

11.1.4 Ex. (210) – Decompose a matrix into multiple matrices based on unique values

```
def matrix_alphabet(t):
    a = []
    n = len(t)
    m = len(t[0])

    for i in range(n):
        for j in range(m):
            q = 1
            for k in range(len(a) + 1):
                if k < len(a) and t[i][j] == a[k]:
                    q = 0
            if q == 1:
                a.append(t[i][j])
    return a


def decompose(c, a):
    n = len(c)
    m = len(c[0])
    d = []

    for i in range(n):
        d.append([])
        for j in range(m):
            d[i].append([])
            for k in range(len(a) + 1):
                d[i][j].append(" ")  # "\u2591"
                if k < len(a) and c[i][j] == a[k]:
                    d[i][j][k] = c[i][j]
    return d
```

Output:

```
M1
 ----------
|111111 111|
|1 1 1 1 11|
|11  1  1 1|
| 1  1 1  1|
|111   1 1 |
|1 111     |
|1         1|
|1 111     |
|11    1   |

  ----------
 M2
  ----------
 |       0    |
 |    0   0   |
 |    0  0 0  |
 |0 0     00  |
 |    0   0 0|
 | 0     0 00|
 | 0      00  |
 | 0     0    |
 |  0000  0   |
  ----------
 M3
  ----------
```

```
    return d

def SMC(m, k):
    r = 'M' + str(k+1)
    r += '\n ----------\n'
    for i in range(len(m)):
        r += "|"
        for j in range(len(m[i])):
            r += str(m[i][j][k])
        r += "|\n"
    r += ' ----------'
    return r

# Main code
r = "u"
c = [
    [1, 1, 1, 1, 1, 1, 0, 1, 1, 1],
    [1, 2, 1, 0, 1, 3, 1, 0, 1, 1],
    [1, 1, 2, 0, 1, 3, 0, 1, 0, 1],
    [0, 1, 0, 2, 1, 3, 1, 0, 0, 1],
    [1, 1, 1, 0, 2, 3, 1, 0, 1, 0],
    [1, 0, 1, 1, 1, 3, 0, r, 0, 0],
    [1, 0, 3, 3, 3, 3, r, 0, 0, 1],
    [1, 0, 1, 1, 1, r, 0, 9, 9, 9],
    [1, 1, 0, 0, 0, 0, 1, 9, 0, 9]
    ]

b = matrix_alphabet(c)
t = decompose(c, b)

for k in range(len(b)):
    print(SMC(t, k))

print(b)
```

```
 ----------
|          |
| 2        |
|  2       |
|   2      |
|    2     |
|          |
|          |
|          |
|          |
 ----------
M4
 ----------
|          |
|     3    |
|     3    |
|     3    |
|     3    |
|     3    |
|  3333    |
|          |
|          |
 ----------
M5
 ----------
|          |
|          |
|          |
|          |
|          |
|       u  |
|      u   |
|     u    |
|          |
 ----------
M6
 ----------
|          |
|          |
|          |
|          |
|          |
|          |
|          |
|       999|
|       9 9|
 ----------
1,0,2,3,u,9
```

```
|          |
|          |
|          |
|          |
|          |
|          |
|          |
|       999|
|       9 9|
 ----------
1,0,2,3,u,9
```

This code implementation from above defines a series of variables and functions to manipulate and decompose a matrix *c* based on an alphabet of unique values found in that matrix. The code also prints the decomposed matrices and the alphabet. The code begins by defining a variable *r* and initializing it with the value "u." Next, there is a 2D array *c*, representing a matrix, where each element is a numerical value. This matrix has dimensions 9×10. The code then calls the *matrix_alphabet* function to extract unique values from the matrix *c* and stores them in an array *b*. It also calls the *decompose* function to decompose the matrix *c* into a 3D array *t*. This decomposition is based on the alphabet found in *b*. A *for-loop* is used to iterate through the elements of the alphabet array *b*, and for each unique value in *b*, it prints a decomposed version of the matrix *t* using the *SMC* function. The *decompose* function takes the matrix *c* and the alphabet array *a* as arguments. It creates a new 3D array *d* and populates it by iterating through the elements of *c*. For each element in *c*, it creates a sub-array in *d*, and for each unique value in *a*, it assigns the corresponding value from *c* to the sub-array in *d*. The *matrix_alphabet* function extracts unique values from the matrix *t* and stores them in an array *a*. It iterates through the elements of *t*, checks if each value is already in *a*, and if not, adds it to *a*. Next, the *SMC* function is used to create a string representation of a decomposed matrix. It takes a matrix *m* and an index *k* representing the alphabet element to consider. It constructs a string *r* that displays the decomposed matrix with the index *k*. The decomposed matrix is surrounded by lines and separators for better visualization. The code concludes by printing the alphabet array *b* and the decomposed matrices for each unique element.

11.1.5 Ex. (211) – Count islands over the matrix and show their location

```
a = [
    [1, 0, 1, 1, 1, 1, 0, 1, 1, 1],
    [1, 0, 1, 0, 1, 1, 0, 0, 1, 1],
    [1, 1, 1, 0, 1, 1, 0, 0, 0, 1],
    [0, 0, 0, 0, 1, 0, 0, 0, 0, 1],
    [1, 1, 1, 0, 1, 1, 1, 0, 1, 0],
    [1, 0, 1, 1, 1, 1, 0, 0, 0, 0],
    [1, 0, 0, 0, 0, 0, 0, 0, 0, 1],
    [1, 0, 1, 1, 1, 1, 0, 1, 1, 1],
```

```
Output:

Islands = 3

2 0 2 2 2 2 0 3 3 3
2 0 2 0 2 2 0 0 3 3
2 2 2 0 2 2 0 0 0 3
0 0 0 0 2 0 0 0 0 3
2 2 2 0 2 2 2 0 3 0
```

```
    [1, 1, 0, 0, 0, 0, 0, 0, 0, 1]
    ]

b = [
    [+1, 0],  # right side element.
    [-1, 0],  # left side element.
    [0, +1],  # upward side element.
    [0, -1],  # downward side element.
    [+1, +1], # upward-right side element.
    [-1, -1], # downward-left side element.
    [+1, -1], # downward-right side element.
    [-1, +1]  # upward-left side element.
    ]

def d(a, i, j, n, m, c):
    if i<0 or j<0 or i>(n-1) or \
       j>(m-1) or a[i][j]!=1:
        return

    if a[i][j] == 1:
        a[i][j] = c + 1

        for k in range(len(b)):
            d(a,i+b[k][0],j+b[k][1],n,m,c)


def SCAN(a):
    n = len(a)    # row.
    m = len(a[0]) # col.
    c = 0         # islands.

    for i in range(n):
        for j in range(m):
            if a[i][j] == 1:
                c += 1
                d(a, i, j, n, m, c)
    return c


def SMC(m):
    r = "\n"
    for i in range(len(m)):
        for j in range(len(m[i])):
            r += str(m[i][j]) + " "
        r += "\n"
    return r

print("Islands =", SCAN(a))
print(SMC(a))
```

```
2 0 2 2 2 2 0 0 0 0
2 0 0 0 0 0 0 0 0 4
2 0 2 2 2 2 0 4 4 4
2 2 0 0 0 0 0 0 0 4
```

This code defines a series of functions and uses them to perform operations on a 2D array called *a*. The array *a* represents a grid or map, containing information about islands and their connectivity. The *a* array is a 2D grid consisting of 10 rows and 10 columns, where each element is either 0 or 1, representing land (1) and water (0). The *b*

array is an array of 2-element arrays, each representing a direction (right, left, upward, downward, etc.) for navigating neighboring elements in the grid. The code then proceeds with the following key functions: *d*, *SCAN*, and *SMC*. Function *d*(*a*, *i*, *j*, *n*, *m*, *c*) is a recursive function that is used to traverse the grid (*a*) starting from a given position (*i*, *j*). It explores neighboring elements and marks connected landmasses with a unique value *c*. This function recursively explores landmasses and increments the *c* value for each new landmass found. Function *SCAN*(*a*) scans the entire grid (*a*) for landmasses (regions of connected land elements). It initializes *c* to zero and, for each land element encountered, increments *c* and calls the *d* function to mark and explore the connected landmass. It returns the total number of landmasses found (value of *c*). Function *SMC*(*m*) is the old, havely used function across this book, that converts a 2D matrix (*m*) into a human-readable string representation. It iterates through the matrix and constructs a string (*r*) where each row of the matrix is separated by a newline character and elements within each row are separated by spaces. In short, the code prints the total number of islands found by calling *SCAN*(*a*) and prints the entire grid *a* in a human-readable format using the *SMC*(*a*) function.

11.1.6 Ex. (212) – Count islands over the matrix and count the characters in each

```
a = [
    [1, 0, 1, 1, 1, 1, 0, 1, 1, 1],
    [1, 0, 1, 0, 1, 1, 0, 0, 1, 1],
    [1, 1, 1, 0, 1, 1, 0, 0, 0, 1],
    [0, 0, 0, 0, 1, 0, 0, 0, 0, 1],
    [1, 1, 1, 0, 1, 1, 1, 0, 1, 0],
    [1, 0, 1, 1, 1, 1, 0, 0, 0, 0],
    [1, 0, 0, 0, 0, 0, 0, 0, 0, 1],
    [1, 0, 1, 1, 1, 1, 0, 1, 1, 1],
    [1, 1, 0, 0, 0, 0, 0, 0, 0, 1]
    ]

b = [
    [1,  0], # right side.
    [-1, 0], # left side.
    [0,  1], # upward side.
    [0, -1], # downward side.
    [1,  1], # upward-right side.
    [-1,-1], # downward-left side.
    [1, -1], # downward-right side.
    [-1, 1]  # upward-left side.
    ]

q = ['*', '#', '%', '&']
p = []

def d(a, i, j, n, m, c):
    if i<0 or j<0 or i>=n or \
```

Output:

```
Islands = 3

* 0 * * * * 0 # # #
* 0 * 0 * * 0 0 # #
* * * 0 * * 0 0 0 #
0 0 0 0 * 0 0 0 0 #
* * * 0 * * * 0 # 0
* 0 * * * * 0 0 0 0
* 0 0 0 0 0 0 0 0 %
* 0 * * * * 0 % % %
* * 0 0 0 0 0 0 0 %

34,8,5
```

```
    j>=m or a[i][j]!=1:
        return

    if a[i][j] == 1:
        a[i][j] = q[c-1]
        p[c - 1] += 1

        for k in b:
            d(a, i+k[0], j+k[1],n,m,c)

def SCAN(a):
    n = len(a)    # row.
    m = len(a[0]) # col.
    c = 0         # islands.

    for i in range(n):
        for j in range(m):
            if a[i][j] == 1:
                c += 1
                p.append(0)
                d(a, i, j, n, m, c)
    return c

def SMC(m):
    r = "\n"
    for row in m:
        for element in row:
            r += str(element) + " "
        r += "\n"
    return r

print("Islands =", SCAN(a))
print(SMC(a))
print(p)
```

Here, the same as in the previous example, the code defines a program to identify and label islands in a binary grid represented by a 2D array *a*. This time, the code also calculates the size of each island and prints the results. The code starts by initializing two 2D arrays, *a* and *b*. Array *a* represents the binary grid, where 1s indicate land and 0s indicate water. Array *b* is an array of pairs used to navigate in all eight possible directions (up, down, left, right, and diagonally) from a given cell. Next, it defines an array *q* containing characters ("*", "#", "%", "&") and an empty array *p*. Note that the code then prints the following: (i) The number of islands found in the grid, which is calculated by the *SCAN* function. (ii) The grid itself with islands labeled by characters from the *q* array. (iii) An array *p* containing the sizes of each island. There are the same three main functions as before. Function $d(a, i, j, n, m, c)$ is a recursive function used to traverse the grid and label the islands. It takes as input the grid *a*, current coordinates (i, j), grid dimensions (n, m), and an island counter *c*. It checks if the cell is out of bounds or not part of an island (1 indicates land), and if so, it returns. Otherwise, it labels the cell, increments the size of the current island in the *p* array, and recursively explores

neighboring cells in all eight directions. Function *SCAN*(*a*) iterates through the entire grid and finds the number of islands. For each unvisited land cell (1), it increments the island counter, initializes the size of the island in the *p* array, and calls the *d* function to label and explore the island. Function *SMC* remains unchanged. The code is now designed to identify and label islands in a binary grid and provides information about the number of islands and their sizes.

11.1.7 Ex. (213) – Count islands and calculate their percentage coverage

```
a = [
    [1, 0, 1, 1, 1, 1, 0, 1, 1, 1],
    [1, 0, 1, 0, 1, 1, 0, 0, 1, 1],
    [1, 1, 1, 0, 1, 1, 0, 0, 0, 1],
    [0, 0, 0, 0, 1, 0, 0, 0, 0, 1],
    [1, 1, 1, 0, 1, 1, 1, 0, 1, 0],
    [1, 0, 1, 1, 1, 1, 0, 0, 0, 0],
    [1, 0, 0, 0, 0, 0, 0, 0, 0, 1],
    [1, 0, 1, 1, 1, 1, 0, 1, 1, 1],
    [1, 1, 0, 0, 0, 0, 0, 0, 0, 1]
    ]

b = [
    [1,  0],  # right.
    [-1, 0],  # left.
    [0,  1],  # upward.
    [0, -1],  # downward.
    [1,  1],  # upward-right.
    [-1,-1],  # downward-left.
    [1, -1],  # downward-right.
    [-1, 1],  # upward-left.
    ]

q = ['*','#','%','&','@','$','!','+','^']
p = [[], [], []]

def d(a, i, j, n, m, c):
    if i<0 or j<0 or i>(n-1) or \
    j>(m-1) or a[i][j]!=1:
        return

    a[i][j] = q[c - 1]
    p[1][c - 1] += 1

    for k in range(len(b)):
        d(a,i+b[k][0],j+b[k][1],n,m,c)

def scan(a):
    n = len(a)     # row.
    m = len(a[0])  # col.
    c = 0          # islands.
```

Output:

```
Islands = 3

* 0 * * * * 0 # # #
* 0 * 0 * * 0 0 # #
* * * 0 * * 0 0 0 #
0 0 0 0 * 0 0 0 0 #
* * * 0 * * * 0 # 0
* 0 * * * * 0 0 0 0
* 0 0 0 0 0 0 0 0 %
* 0 * * * * 0 % % %
* * 0 0 0 0 0 0 0 %

Symbol: *   #   %
Count : 34  8   5
Area  : 38% 9%  6%
```

```
    for i in range(n):
        for j in range(m):
            if a[i][j] == 1:
                c += 1
                p[0].append(q[c - 1])
                p[1].append(0)
                d(a, i, j, n, m, c)
    for i in range(c):

p[2].append(f"{round((100/(n*m))*p[1][i])}%")
    return c

def smc(m, f):
    r = ''
    for i in range(len(m)):
        for j in range(len(m[i])):
            r += str(m[i][j]) + \
                 ps(str(m[i][j]), f)
        r += '\n'
    return r

def ps(a, f):
    t = ''
    b = f - (len(a) % f)
    for i in range(b):
        t += ' '
    return t

print('Islands =', scan(a), '\n')
print(smc(a, 1))
print(smc(p, 4))
```

This latest version of the "island" code adds a couple of new features. It defines a program that processes a 2D array *a* representing a grid of land and identifies and labels islands on the grid. The code begins by defining a 2D array *a*, which represents a grid of land with 1s representing land and 0s representing water. Another 2D array *b* is defined, which represents directional offsets. Each element in *b* is a pair of coordinates that represent movement in different directions (right, left, up, down, etc.). An array *q* is defined, containing various characters used to label the islands on the grid (more than before). An empty 2D array *p* is created to store information about the islands. Next, the code calls the *SCAN*(*a*) function, which scans the grid to identify and label the islands. It then prints the number of islands found and two representations of the grid: the original grid with islands labeled and a grid representation of island information (please see the output above). Inside, the *SCAN*(*a*) function scans the entire grid, calling *d* when it encounters a land element (1). It keeps track of the number of islands found and calculates the percentage of land each island occupies. It returns the total number of islands. Now, the *d*(*a*, *i*, *j*, *n*, *m*, *c*) function is a recursive function used to mark and label islands on the grid. It takes the grid *a*, current coordinates *i* and *j*, grid dimensions *n* and *m*, and a label *c*. It checks if the current position is within bounds and if it is part of an island. If so, it marks the

position with the label from q, updates island information in p, and recursively explores neighboring positions. The $SMC(m, f)$ function is used to convert a 2D array m into a string with proper formatting. It uses the $ps(a, f)$ function to add padding to elements in the matrix for alignment. The $ps(a, f)$ function, like many strategies used here, it was presented in the chapter about functions. Thus, it calculates and returns padding spaces based on the length of the element a and a desired field width f. As a small conclusion, the code effectively identifies and labels islands on the grid, calculates their percentage of coverage, and displays the grid with labels and island information. The significance of this code lies in its practical utility in various applications and its educational value in teaching fundamentals. In order to better understand this example, consider an image that is represented as a matrix a. This image can represent the development of bacterial colonies. Thus, the above algorithm can accurately tell the number of colonies, their area and much more.

11.1.8 Ex. (214) – Show similarities between two strings by sequence alignment

```
# Local sequence alignment
# algorithm and the layout.

def f(a1, a2):
    if a1 == a2:
        return Match
    else:
        return Mismatch

Match = 2
Mismatch = -1
gap = -2

s0 = '1100111111111001'
s1 = '00000011111111100000'

AA = ""
AM = ""
AB = ""

e = ' '

m = []
s = []

MMax = 0
MMin = 0

x = 0
y = 0

# Matrix initialization and completion.
```

Output:

```
  1100111111111001
    ||||||||||||
00000011111111100000
```

```
s = [list(s0), list(s1)]

n_0 = len(s[0]) + 1
n_1 = len(s[1]) + 1

for i in range(n_0):
    m.append([])
    for j in range(n_1):
        m[i].append(0)

        if i == 1 and j > 1:
            m[i][j] = m[i][j-1] + gap
        if j == 1 and i > 1:
            m[i][j] = m[i-1][j] + gap

        if i > 1:
            m[i][0] = s[0][i-2]
        if j > 1:
            m[0][j] = s[1][j-2]

        if i > 1 and j > 1:
            A = m[i-1][j-1]+f(m[i][0],m[0][j])
            B = m[i-1][j]+gap
            C = m[i][j-1]+gap
            D = 0

            m[i][j] = max(A, B, C, D)

            if m[i][j] > MMax:
                MMax = m[i][j]
                x = i
                y = j
            if m[i][j] < MMin:
                MMin = m[i][j]

# Traceback & text alignment.
i = x
j = y

while i >= 2 or j >= 2:
    Ai = m[i][0]
    Bj = m[0][j]

    A = m[i-1][j-1] + f(Ai, Bj)
    B = m[i-1][j] + gap
    C = m[i][j-1] + gap

    if i >= 2 and j >= 2 and m[i][j] == A:
        AA = Ai + AA
        AB = Bj + AB

        if Ai == Bj:
            AM = '|' + AM
```

```
        else:
            AM = e + AM

        i -= 1
        j -= 1

    else:
        if i >= 2 and m[i][j] == B:
            AA = Ai + AA
            AB = '-' + AB
            AM = e + AM
            i -= 1
        else:
            AA = '-' + AA
            AB = Bj + AB
            AM = e + AM
            j -= 1

    r1 = i - 1
    r2 = j - 1

    if m[i][j] <= 0:
        break

# Layout.
tM = ''
tS = ''

# Check the end.
AA += s0[x-1:n_0-x]
AB += s1[y-1:n_1-y]

# Check the beginning.
AA = s0[:r1] + AA
AB = s1[:r2] + AB

if r1 > r2:
    v = r1 - r2
    tS += e * v
    tM += e * (v + r2)
    AB = tS + AB
    AM = tM + AM
else:
    v = r2 - r1
    tS += e * v
    tM += e * (v + r1)
    AA = tS + AA
    AM = tM + AM

# Print the alignment.
print(AA)
print(AM)
print(AB)
```

The above code performs sequence alignment using a dynamic programming approach, specifically for pairwise sequence alignment of two strings [25]. The code begins by defining several variables to be used in the sequence alignment algorithm. These variables include *Match*, *Mismatch*, and *gap* penalties, two input sequences ($s0$ and $s1$), and three strings (*AA*, *AM*, and *AB*) to store the aligned sequences and matching characters. Note that *AA* means alignment of sequence A, *AM* means the alignment of sequence in the middle (the connection vertical lines between the characters of A and B), and *AB* means alignment of sequence B. Additionally, the code initializes an empty string *e* for placeholder characters, and two empty arrays *m* and *s* for matrices used in the alignment calculations. It also sets variables *MMax* and *MMin* to store the maximum and minimum values in the alignment matrix, and initializes *x* and *y* to track their positions. The next section of the code initializes and completes the alignment matrix *m*. It sets up a nested loop to iterate through the matrix, calculating alignment scores based on the dynamic programming algorithm. It uses the variables *A*, *B*, *C*, and *D* to calculate the maximum alignment score at each cell of the matrix. The code keeps track of the maximum and minimum scores, as well as their corresponding positions in the matrix (*x* and *y*). After completing the matrix, the code proceeds to perform traceback to find the aligned sequences. It starts from the position (*x*, *y*) with the maximum score and traces back through the matrix, building the aligned sequences and using the *Match*, *Mismatch*, and *gap* penalties as needed. Next, the code adjusts the layout of the aligned sequences, ensuring they have the same length and align correctly. It adds placeholder characters to the beginning and end of the sequences if needed to align them properly. The final section of the code prints the aligned sequences (*AA*, *AM*, and *AB*) to the console, representing the aligned sequences, matching characters, and gaps. Additionally, there is a matching function $f(a1, a2)$ defined in the code, which returns a *Match* or *Mismatch* penalty based on whether two characters are equal or not. Overall, this code performs sequence alignment between two input sequences and prints the aligned sequences along with matching characters and gaps. The provided Python code has several practical applications in bioinformatics, computational biology, and related fields. For example, in the field of Genetics, sequence alignment is commonly used to compare DNA, RNA, or protein sequences to identify similarities or differences. This is important in understanding evolutionary relationships, identifying functional elements in genomes, and annotating genes. Also, sequence alignment is crucial for searching biological databases, such as *GenBank* or *UniProt*, to find sequences similar to a query sequence. This is often used to identify potential homologous genes or proteins. Nonetheless, sequence alignment has many applications, from biology to antivirus engines to finding commonalities between multiple files infected with polymorphic viruses, and the cases for applications go on as a function of need and imagination.

Randomnes and Programming

12

Randomness plays a significant role in various aspects of computer programming, and Python, as a versatile computer language, offers powerful tools and techniques to incorporate randomness into software applications [26]. Randomness refers to the concept of unpredictability and uncertainty, essential for tasks like generating random numbers, shuffling data, simulating random events, or creating games and simulations [26]. In Python, developers can use this unpredictability via built-in functions and libraries designed to handle randomization effectively [1]. However, this chapter will make an introduction into the significance of randomness in computer programming, its applications, and how Python empowers developers to implement applications, enhancing the functionality and reliability of software.

P. A. Gagniuc, *Coding Examples from Simple to Complex*, Synthesis Lectures on Computer Science, https://doi.org/10.1007/978-3-031-53812-4_12

12.1.1 Ex. (215) – Get complementary array by using random values

```
# understand randomness - get
# complementary array by using
# random values.

import random

def mutate(a):
    m = len(a)
    n = 200
    b = [0] * m

    for i in range(n):
        s = 0  # score.

        for j in range(m):
            b[j] = round(random.random())

            # complementary array.
            if b[j] != a[j]:
                s += 1

            # identic array
            # if b[j] == a[j]:
            #     s += 1

        if s >= m:
            return b

    return "not found by random means."

a = [1, 0, 0, 1, 1, 1, 0]
print(mutate(a))
```

```
Output:

0,1,1,0,0,0,1
```

The code from above aims to demonstrate randomness and create a complementary array through random values. It starts by defining two arrays, *a* and *b*, with some initial values. The *mutate* function is defined to mutate the array *a* into a complementary array *b* using random values. Inside the function, the code sets *m* as the length of array *a* and variable *n* as value 200. It then initializes a variable *s* (score) to keep track of how many elements in the *b* array differ from the corresponding elements in the *a* array. It then enters a loop that runs *n* times, attempting to create a complementary array. Within this loop,

it resets the score *s* to 0 for each iteration. Inside the nested loop, it generates random values (0 or 1) using the *random* function (from the *random* library) for each element of the *b* array. If the generated value is different from the corresponding element in the array *a*, it increments the score *s*, indicating a difference between the arrays. After generating a random *b* array, it checks if the score *s* is greater than or equal to the length *m* of the array *a*. If this condition is met, it returns the complementary array *b*. If the condition is not met after *n* iterations, it returns the message "not found by random means." Outside the function, the source code prints the result of calling *mutate*(*a*).

12.1.2 Ex. (216) – Take the first 20% of the closest solutions (mutation/selection)

```
# understand randomness,
# take the first 20% of
# the closest solutions.

import random

a = [1, 0, 0, 1, 1, 1, 0]
b = [0, 0, 0, 0, 0, 0, 0]

# Initialize c with a maximum
# possible size based on n and m.
# The size of c will be reduced
# later to match the actual number
# of elements.

n = 100
m = len(a)
c = [[None for _ in range(m + 1)] \
      for _ in range(n)]

def mutate(c):
    p = 20  # Select best X%.
    q = round(m * (1-p / 100))
    k = 0

    for i in range(n):
        s = 0

        for j in range(m):
```

```
Output:

1 0 1 1 1 1 0 6
0 0 0 1 1 1 0 6
1 0 1 1 1 1 0 6
0 0 0 1 1 1 0 6
1 0 0 1 1 1 1 6
1 0 0 1 1 1 0 7
1 1 0 1 1 1 0 6
1 0 1 1 1 1 0 6
1 1 0 0 0 1 0
```

```
            b[j] = round(random.random())
            if b[j] == a[j]:
                s += 1
            c[k][j] = b[j]

        if s >= q:
            c[k][m] = s
            k += 1

    # Reduce the size of c to match
    # the actual number of elements.

    return c[:k]

def SMC(m):
    r = ''
    for row in m:
        r += ' '.join(str(x) for x in row) + "\n"
    return r

fit = mutate(c)
print(SMC(fit))
```

This code aims to demonstrate a process of mutation and selection within an algorithm. The code begins by defining two arrays, *a* and *b*, representing binary sequences. Variable *a* contains a binary sequence [1,0,0,1,1,1,0], and *b* is initialized as an array of zeros of the same length. An array *c* is declared, which will be used to store mutated sequences. The *mutate*(*c*) function is defined to perform mutation and selection. Inside this function a series of events unfold. The length of array *a* is stored in the variable *m*, and the value 100 is stored in *n*. Variable *k* is initialized to 0, and *p* is set to 20, representing the percentage of best solutions to select. Variable *q* is calculated as the number of elements to select, which is *m* multiplied by $(1 - p/100)$. Next, there is a loop that runs *n* times, where *n* is the number of mutations to be performed, namely: Variable *s* is reset to 0, and a sub-array is initialized in the *c* array at index *k*. Another loop iterates through the elements of array *a* (length *m*). For each element, a random binary value is generated and stored in the corresponding position of array *b*. If the generated value matches the value in array *a* at the same position, *s* is incremented, and the value is stored in the corresponding position in the sub-array of *c*. If the value of *s* (number of matches with *a*) is greater than or equal to *q*, the sub-array in *c* also stores the value of *s*, and *k* is incremented. The *mutate*(*c*) function returns the mutated and selected sequences stored in array *c*. Additionally, there is the *SMC*(*m*) function that takes a matrix (*m*) as an argument and converts it into a human-readable string format. It iterates through the rows and columns of the matrix, appending each element followed by a space and adding a newline character after each row. Then, the result *r* is returned. Thus, the code calls the *mutate*(c) function, stores the result in the variable *fit*, and prints the result of converting *fit* into a string format using the *SMC*(*fit*) function.

12.1.3 Ex. (217) – Find complementary matrix by using stochastic means

```
# Find complementary matrix
# by using stochastic means
# (uniform distribution).

import random

def mutate(a):
    n = len(a)
    m = len(a[0])
    q = 200
    s = 0  # score.

    for k in range(q):
        s = 0
        b = []

        for i in range(n):
            b.append([])

            for j in range(m):
                b[i].append(round(random.random()))
                if b[i][j] != a[i][j]:
                    s += 1

            if s >= m * n:
                return b

    return "not found by random means."

def smc(m):
    if isinstance(m, str):
        return m
    r = ''
    for i in range(len(m)):
        for j in range(len(m[i])):
            r += str(m[i][j]) + " "
        r += "\n"
    return r

a = [
    [0, 0, 0],
    [0, 1, 0],
    [0, 0, 0],
    ]

print(smc(mutate(a)))
```

Output:

```
1 1 1
1 0 1
1 1 1
```

This current example aims to find a complementary matrix to a given matrix *a* using stochastic means (uniform distribution). The code begins with the definition of matrix *a*, which is a 3×3 matrix filled with numeric values, and an empty matrix *b*. Next, it calls the *SMC* function and passes the result of the *mutate*(*a*) function as an argument to print the matrix obtained after the mutation. The *mutate*(*a*) function is defined to find a complementary matrix through a stochastic process. It takes the matrix *a* as an input and performs the following steps: (i) It calculates the dimensions of matrix *a* by getting the number of rows (*n*) and columns (*m*). (ii) It sets a variable *q* to 200, which represents the number of attempts to find a complementary matrix. (iii) It initializes variable *s* to 0, which will be used to keep track of the score (the number of differences) between the original matrix *a* and the mutated matrix *b*. (iv) It enters a loop that runs for *q* iterations. In each iteration, it resets the score *s* to 0. Inside the loop, it iterates through each element of the matrix *a* using nested for loops. For each element, it generates a random number between 0 and 1 by using the *random*() function (import *random*), then it rounds it to the nearest integer using the *round*() function, and it assigns the result to the corresponding element in matrix *b*. It also checks if the value in *b* differs from the value in *a* and increments the score *s* if there is a difference. After completing the iteration over all elements of the matrix, it checks if *s* is greater than or equal to the total number of elements in the matrix ($m \times n$). If this condition is met, it returns the mutated matrix *b*, which is now complementary to *a*. If the loop finishes all iterations and no complementary matrix is found, it returns the string "not found by random means." The *SMC* function is defined to format matrix *m* as a string with rows separated by newline characters. It iterates through the matrix and builds a string representation of the matrix. In short, the code attempts to find a complementary matrix to the input matrix *a* by randomly mutating its values and checking if the score of differences reaches a certain threshold. If a complementary matrix is found, it is returned; otherwise, a message of not found is shown.

12.1.4 Ex. (218) – A two states *Markov Chain* simulator based on letters

```
import random

Jar = ["WWBBBBBBBB", "WWWWWBBBBB"]

draws = 17
z = ''

def Draw(S):

    # Choose a random character
    # from the string at Jar[S].

    rc = random.randint(0, len(Jar[S]) - 1)
    ball = Jar[S][rc]
    return ball

# Initial draw
a = Draw(1)
z += f" Jar W[{a}],"

for i in range(1, draws + 1):
    if a == "W":
        a = Draw(0)
        z += f" Jar W[{a}],"
    else:
        a = Draw(1)
        z += f" Jar B[{a}],"

print(z)
```

Output:

```
Jar W[B],
Jar B[B],
Jar B[W],
Jar W[W],
Jar W[B],
Jar B[W],
Jar W[B],
Jar B[W],
Jar W[B],
```

The point of this code is to demonstrate how to simulate a non-uniform random process based on quantities (drawing balls from jars) and keep track of the results [26]. It showcases the use of functions, loops, and conditional statements to achieve this simulation. In this code, there is an array called *Jar* with two elements at indices 0 and 1, representing jars filled with colored balls. The strings in these array elements represent the content of the jars. The variable *draws* is set to 17, which indicates the number of draws to be performed. The variable z is initialized as an empty string. The code defines a function called $Draw(S)$ that takes a parameter S, which is used to choose a jar (0 for the first jar and 1 for the second jar). Inside this function it initializes a variable rc to 0 (which stands for *randomly_choose*). Then, it generates a random floating-point number between 0 and 1 using *randint* function (from the *random* library) and multiplies it by the length of the jar specified by S. This number is stored in rc, effectively pointing to an index in the jar. Next, the code extracts a single ball (character) from the jar specified by S at the chosen index ($ball = Jar[S][rc]$) and returns it. Outside the function, there is a loop that iterates from $i = 1$ to *draws* (17 times). Inside this loop it checks whether the value of a is equal to "W." If it is, it calls the *Draw* function with S as 0 (indicating the first jar) and appends the result to the string z, including the label "Jar W[]". If a is

not equal to "W," it calls the *Draw* function with *S* as 1 (indicating the second jar) and appends the result to the string *z*, including the label "Jar B[]". Next and last, the code prints the accumulated string *z* to the console for inspection.

12.1.5 Ex. (219) – A two states *Markov Chain* simulator based on probability values

```
import random

draws = 8
Jar = ["", ""]
z = ''

def Fill_Jar(S, p):
    Balls_W = round(100 * p)
    Balls_B = 100 - Balls_W

    for i in range(1, Balls_W + 1):
        Jar[S] += "W"
    for i in range(1, Balls_B + 1):
        Jar[S] += "B"
```

Output:

```
Jar W[B],
Jar B[W],
Jar W[B],
Jar B[W],
Jar W[B],
Jar B[W],
Jar W[B],
Jar B[W],
Jar W[B],
```

```
def Draw(S):
    rc = random.randint(0, len(Jar[S]) - 1)
    ball = Jar[S][rc]
    return ball

Fill_Jar(0, 0.2)
Fill_Jar(1, 0.6)

a = Draw(1)
z += f" Jar W[{a}],"

for i in range(1, draws + 1):
    if a == "W":
        a = Draw(0)
        z += f" Jar W[{a}],"
    else:
        a = Draw(1)
        z += f" Jar B[{a}],"

print(z)
```

In this code, there is a simulation of drawing balls from two jars, each containing white ("W") and black ("B") balls. In other words, it is the same non-uniform random process as shown previously, but this time, instead of a sequence of objects (characters) that sets the probability distribution, the implementation now uses transition probability values to perform the simulation. The code begins with the initialization of variables and the declaration of two functions. The *draws* variable is set to 8, and an empty string *z* is defined. An array *Jar* is created with two elements, initialized as empty strings. The *Fill_Jar* function is defined to populate the jars. It takes two arguments: *S* (representing the jar number) and *p* (representing the probability of drawing a white ball from the jar). It calculates the number of white and black balls based on the probability *p* and fills the jar accordingly. Next, the code draws the first ball (*a*) from the second jar (*Jar*[1]) using the *Draw* function and appends the result to the string *z*. A loop runs for *draws* times, simulating subsequent ball draws. In each iteration, it checks the color of the last drawn ball (*a*). If it is white ("W"), the code draws a ball from the first jar (*Jar*[0]), and if it is black ("B"), the code draws a ball from the second jar (*Jar*[1]). The results of these draws are appended to the string *z*. Once the execution of the loop finishes, the code prints the string *z* which contains a sequence of ball draws from both jars, indicating whether they are white ("W") or black ("B"). Two additional functions, *Draw* and *Fill_Jar*, are defined to facilitate the simulation. The *Draw* function takes a parameter *S* (indicating the jar) and randomly selects a ball from that jar based on its length and returns it. The *Fill_Jar* function takes parameters *S* and *p*, calculates the number of white and black balls based on the probability *p*, and fills the specified jar accordingly. The point of the provided Python code is to simulate a random process of drawing balls from two jars, each containing white ("W") and black ("B") balls, that is, two states. The code allows the reader to specify the probability of drawing a white ball from each jar and then simulates a series of ball draws based on these probabilities. Thus, the main purpose of the code is to demonstrate a simple stochastic process and record the outcomes of multiple ball draws from the two jars. It creates a string *z* that represents a sequence of ball draws, with each draw being either "W" or "B" depending on the color of the ball drawn from the corresponding jar. The code could be used for various purposes, because it provides a basic framework for simulating and recording random events, making it a useful tool for educational, experimental, or illustrative purposes related to probability and randomness.

12.1.6 Ex. (220) – Multiply a probability vector with a probability matrix n times

```
a = [
    [1.0, 0.0, 0.0, 0.0],
    [0.5, 0.0, 0.5, 0.0],
    [0.0, 0.5, 0.0, 0.5],
    [0.0, 0.0, 1.0, 0.0]
    ]

v = [ [0, 0, 0, 1],
      [0, 0, 0, 0],]

c = 5
n = len(a)
m = len(a[0])

for k in range(1, c + 1):

    for i in range(n):
        for j in range(m):
            v[1][j] += v[0][i] * a[i][j]

    for i in range(m):
        v[0][i] = v[1][i]
        v[1][i] = 0

        print(
        f'k({k})=['
        f'{v[0][0]},'
        f'{v[0][1]},'
        f'{v[0][2]},'
        f'{v[0][3]}]'
             )
```

Output:

```
k(1)=[0,0,1,0]
k(2)=[0,0.5,0,0.5]
k(3)=[0.25,0,0.75,0]
k(4)=[0.25,0.375,0,0.375]
k(5)=[0.44,0,0.56,0]
```

The primary use of this code is related to predictions. Thus, it repeatedly performs matrix–vector multiplication c times, accumulating the results in v, and displaying the updated v vector after each iteration. This could be part of a numerical computation or simulation where iterative updates to a vector are necessary, such as in some mathematical or scientific simulations of *Markov Chains*. The code starts by defining two matrices, a and v, as well as two scalar variables, c, and n and m. Matrix a is a 4×4 matrix with specific numerical values. Matrix v is a 2×4 matrix initialized with zeros. Scalar c is set to 5, representing the number of iterations in the subsequent loop. Scalar n is assigned the value of the number of rows in matrix a (which is 4) whereas scalar m is assigned the value of the number of columns in matrix a (which is 4). The code then enters a nested loop structure. There are two outer loops controlled by the variable k, which ranges from 1 to c. These loops are responsible for performing matrix–vector multiplications. Within the loop, there are nested loops controlled by i and j, which iterate through the rows and columns of matrices a and v. Inside these loops, the code performs calculations to update the values in the v matrix based on matrix multiplication between a and v. After each iteration of the k loop, there is a block of code that updates the v matrix

for the next iteration. The values in $v[0]$ are updated with the values calculated in the previous iteration, and $v[1]$ is reset to zeros. Next, the code prints the result of each iteration, showing the values of $v[0]$ in a formatted string. Overall, this implementation is performing iterative matrix–vector multiplications c times and displaying the results for each iteration.

12.1.7 Ex. (221) – A *Markov Chain* framework for simulation

```
import random

def Draw(S):
    rc = random.random() * len(Jar[S])
    ball = Jar[S][int(rc)]
    return ball

def Fill_Jar(S):
    Ltot = 10
    b = ""
    for i in range(m):
        # Check if the element is a number.
        if isinstance(P[S][i], (int, float)):
            k = round(Ltot * P[S][i])
            for j in range(k):
                b += P[0][i]
    return b

def SMC(m):
    r = ''
    for i in range(len(m)):
        for j in range(len(m[i])):
            r += m[i][j]
        r += "\n"
    return r

P = [
    ["A", "B", "C", "D"],
    [0.00, 0.50, 0.50, 0.00],
    [0.33, 0.00, 0.33, 0.33],
    [0.00, 1.00, 0.00, 0.00],
```

Output:

```
Q = CBCBCBCBDC

BBBBBCCCCC
AAACCCDDD
BBBBBBBBB
CCCCCCCCC
```

```
    [0.00, 0.00, 1.00, 0.00]
    ]

n = len(P)    # n = 5.
m = len(P[0]) # m = 4.

Jar = []

for j in range(m+1):
    Jar.append(Fill_Jar(j))

draws = 10
a = Draw(1)  # first draw.
q = ""

for i in range(1, draws+1):
    for j in range(m):
        if a == P[0][j]:
            a = Draw(j + 1)
            q += P[0][j]
            break

print("Q = " + q)
print(SMC(Jar))
```

The purpose here is to create a simulation of drawing items from a jar with predefined probabilities and capture the sequence of draws. It demonstrates how to model a random process and calculate outcomes based on specified probabilities. The code starts by defining a two-dimensional array P, representing a probability distribution for drawing items from the jar. It contains items labeled as "A," "B," "C," and "D," along with associated probabilities. Next, it calculates the number of rows (n) and columns (m) in the P array, and it initializes an empty array *Jar*. A variable *draws* is set to 10, indicating the number of draws to be simulated. The code initializes variables a, q, and z for tracking the draws. A loop iterates over the number of draws specified. Within this loop, another loop iterates over the columns of the P array to determine which item is drawn based on the probabilities. The selected item is added to the q variable, which records the sequence of draws. There is a *Draw*(S) function that simulates drawing an item based on probabilities specified in a specific row of the P array. It randomly selects an item from the jar and returns it. Another function *Fill_Jar*(S) is defined to fill the jar based on probabilities from a specific row of the P array. It calculates the number of items to add to the jar for each item type based on the probabilities. The code then prints the contents of the *Jar* array by using the *SMC*(m) function. Thus, the code simulates drawing items from a jar based on specified probabilities and records the sequence of draws in the q variable. One last thing to note, is that the above example, is the most primitive version of a *GPT*-like (generative pre-trained transformer) system. Once the reader carefully understands the principles, it will get the point of the example by looking at the output.

Python Specific

13

Python is a versatile computer language and an invaluable tool for programmers and data scientists alike [1]. Up to this point all examples were general and computable in all imperative computer languages. However, there are methods that are mainly language specific and a few of such examples are shown here. One of its many strengths lies in handling various forms of data encoding and decoding, such as Base64, which is effortlessly managed through built-in functions. This feature is particularly useful for data transmission and encoding media files into text formats. Furthermore, Python excels in working with JSON, a popular data format used for configuration files and data interchange on the web. Python can convert JSON data to and from files, allowing for efficient storage and retrieval of structured data. This functionality makes Python an excellent choice for web development and data analysis tasks. Another area where Python shines is file I/O operations. It provides straightforward mechanisms for reading from and writing to files, making it an easy-to-use computer language for file manipulation tasks (whether it is reading log files, writing data to CSV files, or processing text files) [27]. Python also offers robust capabilities for data visualization through various charting libraries. These libraries enable the creation of a wide range of charts and graphs, from simple line graphs to complex heatmaps, aiding in the analysis and presentation of data. The language is also well-equipped for developing desktop applications with graphical user interfaces (GUIs). Python enables the creation of user-friendly interfaces for applications by using libraries like *Tkinter*, thus, making it accessible to non-programmers and enhancing the interactivity of Python scripts. Lastly, one important capability is the handling of HTML requests, which is a must for working with data stored remotely on the Internet. With libraries like *requests*, Python is adept at interacting with the web, fetching and processing web content. This is particularly useful for web scraping, API interactions, and automating

P. A. Gagniuc, *Coding Examples from Simple to Complex*, Synthesis Lectures on Computer Science, https://doi.org/10.1007/978-3-031-53812-4_13

web-based tasks. Therefore, this chapter explores the wide array of "must have" functionalities, ranging from Base64 encoding/decoding, JSON processing, file I/O, charts, GUI development, to handling HTML requests.

13.1.1 Ex. (222) – *Base64* encoding and decoding via built-in functions

```
import base64

ori = "this is a text!"

# Encode a string.
enc = base64.b64encode(ori.encode()).decode()

# Decode the string.
dec = base64.b64decode(enc).decode()

print(enc)
print(dec)
```

Output:

```
dGhpcyBpcyBhIHRleHQh
this is a text!
```

Base64 encoding is and will likely be indispensable for web applications that must be independent of external files. For instance, many file types can be embedded as text in an HTML file via Base64 encoding. Thus, within the above script, there are several operations performed on strings using the *b64encode* and *b64decode* functions. First, the python library called base64 is imported, and a variable *ori* is defined and initialized with the string "this is a text!" Then, the *b64encode* function is used to encode the *ori* string into a Base64-encoded string, and the result is stored in a variable called *enc*. Next, the *b64decode* function is used to decode the Base64-encoded string stored in *enc*, and the result is stored in a variable called *dec*. After these encoding and decoding operations, the code displays two lines in the output. The first line displays the Base64-encoded string (*enc*), and the second line displays the decoded string (*dec*). Thus, this code encodes a string using Base64 encoding and then decodes it back to its original form, demonstrating how to use the *b64encode* and *b64decode* functions for encoding and decoding strings in Python.

13.1.2 Ex. (223) – Local storage

```
import json

# any object to string.

a = ["a", "b", "c"]

b = [
        [0, 1, 0],
        [1, 1, 1],
        [0, 1, 0]
    ]

c = {"c1": a, "c2": b, "c3": 42}

obj = {"v1": a, "v2": b, "v3": c}

# Store data:
txt = json.dumps(obj)
with open('storage.txt', 'w') as file:
    file.write(txt)

# Retrieve data:
with open('storage.txt', 'r') as file:
    txt = file.read()
obj = json.loads(txt)

msg = ''
msg += 'From string:\n' + txt
msg += '\n\nFrom object:\n' + obj['v1'][1]
msg += '\n' + str(obj['v2'][0][1])
msg += '\n' + str(obj['v3']['c2'])
msg += '\n' + str(obj['v3']['c2'][1][1])

print(msg)
```

Output:

```
From string:
-----------------
{"v1":["a","b",
"c"],"v2":[[0,1,0],
[1,1,1],[0,1,0]],
"v3":{"c1":["a",
"b","c"],"c2":
[[0,1,0],[1,1,1],
[0,1,0]],"c3":42}}

From object:
-----------------
b
1
0,1,0,1,1,1,0,1,0
1
```

This code performs various operations related to data serialization and storage using text files. The code defines several variables: (i) Variable *a* is an array containing the elements "a," "b," and "c." (ii) Variable *b* is a nested array representing a 3×3 matrix with binary values. (iii) Variable *c* is an object with three key-value pairs: "c1", "c2", and "c3", where "c1" is assigned the *a* array, "c2" is assigned the *b* matrix, and "c3" is assigned the number 42. Also, there is variable *obj* that is an object with three key-value pairs: "v1", "v2", and "v3", where "v1" is assigned the a array, "v2" is assigned the *b* matrix, and "v3" is assigned the *c* object. The code proceeds to store the *obj* object as a JSON string (*json.dumps* function) inside a text file (i.e., *open* function). Next, it retrieves the stored JSON string from the same text file. Then it parses it back into a Python object, assigning it to the *obj* variable. The code constructs a message (*msg*) by concatenating strings and data from the *obj* object. It includes both the original JSON string and specific values from the parsed object, such as elements from *v1*, an element from *v2*, the entire

c2 array, and an element from *c2*. These values are appended to the *msg* string with "\n" line breaks for formatting. At the end, it displays the constructed *msg*, effectively showing the serialized data and the parsed values.

13.1.3 Ex. (224) – File I/O

```
# read a file.

def read_file(fp):
    try:
        with open(file_path, 'r') as file:
            content = file.read()
            print(content)
    except FileNotFoundError:
        print("File not found.")

read_file('path/file.txt')


# write file

def write_file(file_path, text):
    with open(file_path, 'w') as file:
        file.write(text)
    print("Content written to file.")

write_file('path/file.txt', 'This is data.')
```

Output:

```
This is data.
```

The first part of the code defines a function *read_file* which is designed to read and print the content of a file. The function takes one argument, *fp*, which is intended to be the file path of the file to be read. Inside the function, a *try* block of code is used to attempt to open the specified file in read mode ('r'). If the file is found and successfully opened, its content is read using *file.read*() and then printed. In case the file is not found, a *FileNotFoundError* is caught by the *except* block, and a message "File not found." is printed. This function is then called with the argument "path/file.txt", implying that it attempts to read a file located at this path. The second part of the code defines another

function *write_file*, which takes two arguments: *file_path* and *text*. Variable *file_path* is the location where the file should be created or overwritten, and *text* is the content that will be written into the file. This function opens the specified file in write mode ('w'). When a file is opened in write mode, any existing content in the file is erased, and the new content is written from the start of the file. The text is written into the file using *file.write*(*text*). After writing the content, a confirmation message "Content written to file." is printed. The function is then called with the arguments "path/file.txt" and "This is data.", meaning it creates or overwrites a file at that path with the text "This is data." Thus, these two functions demonstrate basic file operations in Python: reading from and writing to files. The *read_file* function is designed to handle the case where the specified file might not exist, preventing the program from crashing by using a *try-except* block. The *write_file* function showcases how to overwrite or create a new file with given content.

13.1.4 Ex. (225) – Charts

```
import tkinter as tk

def draw_chart(canvas, q, c, e):
    s = [int(i) for i in q.split(',')]
    max_value = max(s)
    width = canvas.winfo_reqwidth()
    height = canvas.winfo_reqheight()

    if e == 'y':
        canvas.delete("all")

    canvas.create_rectangle(0, 0, width, height, fill="#f1f1f1")

    prev_x = prev_y = None
    for i in range(len(s)):
        y = height - (height / max_value * s[i])
        x = (width / len(s)) * i
        if prev_x is not None and prev_y is not None:
            canvas.create_line(prev_x, prev_y, x, y, fill=c, width=2)
        prev_x, prev_y = x, y

def create_chart_window(q, c, e):
    root = tk.Tk()
    root.title("Chart")
```

```
    canvas = tk.Canvas(root, height=300, width=1100)
    canvas.pack()
    draw_chart(canvas, q, c, e)
    root.mainloop()

create_chart_window('23,45,66,77,44,33,99', '#ff0000', 'y')


# or a second version:


import matplotlib.pyplot as plt

def chart(q, c, e):
    s = [int(val) for val in q.split(",")]
    mx = max(s)

    # Size 1100x300 pixels.
    plt.figure(figsize=(11, 3))

    if e == 'y':
        # Clear the current figure.
        plt.clf()

    # Set the background color.
    plt.gca().set_facecolor("#f1f1f1")

    x = range(len(s))
    # Invert the values.
    y = [mx - val for val in s]

    plt.plot(x, y, color=c, linewidth=2)

    # Set the limits of y-axis.
    plt.ylim(0, mx)

    plt.show()

chart('23,45,66,77,44,33,99', '#ff0000', 'y')
```

Output:

The first script provided uses the *tkinter* library in Python to draw a chart, while the second script uses the *matplotlib* library for the same purpose. Both scripts visualize a

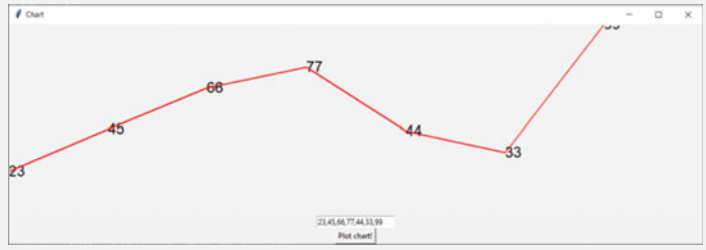
series of numerical values as a line chart, but they approach the task differently due to the nature of the libraries used. In the *tkinter* script, a function *draw_chart* is defined to create the chart. It takes a canvas from the *tkinter* GUI toolkit, a string *q* containing comma-separated values, a color *c* for the chart line, and a flag *e* that determines whether the canvas should be cleared before drawing. The string of numbers is split and converted into integers, and the maximum value is determined to scale the chart correctly. The width and height of the chart are obtained from the canvas properties. A rectangle with a light grey background (#f1f1f1) is created as the background of the chart. The script then iterates over the data points, plotting each as a segment of the line and connecting them. The color and width of the line are customizable. The *create_chart_window* function sets up the *tkinter* window, creating a canvas of specified size and invoking *draw_chart* to draw the chart on this canvas. The script concludes by creating a window with the *create_chart_window* function, passing the data string, color, and erase flag. On the other hand, the *matplotlib* script defines a *chart* function that also takes the same three parameters: the data string, color, and erase flag. Here, the data is processed similarly, and a figure with a set size is created using *matplotlib*. If the erase flag is "y", the current figure is cleared. The background color of the chart is set, and the data is plotted with the specified color and line width. The *y*-axis is inverted and limited based on the maximum value in the data. Next, the chart is displayed using *plt.show*(). Note that both scripts demonstrate different methods of creating simple line charts in Python, with *tkinter* providing more control over the GUI aspects, while *matplotlib* offers a more straightforward approach for creating and displaying plots.

13.1.5 Ex. (226) – The Graphical User Interface (GUI)

```
import tkinter as tk

def draw_chart(canvas, q, c, e):
    s = [int(i) for i in q.split(',')]
    max_value = max(s)
    width = canvas.winfo_reqwidth()
    height = canvas.winfo_reqheight()

    if e == 'y':
        canvas.delete("all")

    canvas.create_rectangle(0, 0, width, height, fill="#f1f1f1")

    prev_x = prev_y = None
    for i in range(len(s)):

        y = height - (height / max_value * s[i])
        x = (width / len(s)) * i
```

```
        canvas.create_text(x, y, text=str(s[i]),
        font=("Arial", 18), fill="black", anchor='w')

        if prev_x is not None and prev_y is not None:
            canvas.create_line(prev_x, prev_y, x, y, fill=c, width=2)
        prev_x, prev_y = x, y

def create_chart_window():
    root = tk.Tk()
    root.title("Chart")

    canvas = tk.Canvas(root, height=300, width=1100)
    canvas.pack()

    entry = tk.Entry(root)
    entry.insert(0, "23,45,66,77,44,33,99")
    entry.pack()

    def plot_chart():
        data = entry.get()
        draw_chart(canvas, data, '#ff0000', 'y')

    button = tk.Button(root, text="Plot chart!", command=plot_chart)
    button.pack()

    root.mainloop()

create_chart_window()
```

Output:



The given Python script uses the *Tkinter* library to create a simple graphical user interface (GUI) for drawing a line chart based on user input. As specified in the previous example, *Tkinter* is a standard GUI toolkit in Python, used for creating simple and effective GUI applications. The source code defines two main functions, namely: *draw_chart* and *create_chart_window*. The *draw_chart* function is mainly responsible for drawing the chart on a *Tkinter* canvas widget. Much like before in the previous example, it takes four parameters: *canvas* (the canvas on which to draw), *q* (a string of comma-separated values representing the data points), *c* (the color of the line in the chart), and *e* (a flag to erase the previous drawing). The function begins by converting the string of data points (*q*) into

a list of integers (*s*). It then determines the maximum value in this list to scale the chart appropriately. The width and height of the canvas are acquired using *winfo_reqwidth*() and *winfo_reqheight*() methods. If the *e* parameter is "y", it clears the canvas to ensure that the new chart does not overlap with any previous drawings. The background of the canvas is set to a light gray color. The function then iterates over the data points, calculating the *x* and *y* coordinates for each point based on the canvas dimensions and the value of the data point. Each data point value is displayed as text on the canvas. Next, a line is drawn between consecutive data points to create the line chart. The color and width of the line are determined by the *c* parameter and a fixed value, respectively. The *create_chart_window* function sets up the *Tkinter* window. It creates the main window (root), sets its title, and initializes a canvas widget with specific dimensions. An entry widget is also created for users to input the data points as a comma-separated string. A button is provided to trigger the chart drawing. When clicked, it fetches the data from the entry widget and calls *draw_chart* to plot the chart. The script concludes with an example usage, calling *create_chart_window* to start the application. This function call initializes the *Tkinter* loop, displaying a window where users can enter data points and view the corresponding line chart.

13.1.6 Ex. (227) – HTTP requests

```
import requests

url = 'https://www.springer.com'

# Send a GET request to the URL.
response = requests.get(url)

# Check if the request
# was successful.

if response.status_code == 200:

    # Get the HTML content
    # of the page.

    html_content = response.text
    print(html_content)
else:
    print("Failed to retrieve the webpage")


# or a second version:
```

```python
import requests

url = 'https://springer.com/api'

# Make a dictionary
# containing query
# parameters.

params = {
'param1': 'value1',
'param2': 'value2'
}

# Send a GET request
# with query parameters.

response = requests.get(url, params=params)

# Check if the request was successful.
if response.status_code == 200:

    # Process the response if needed.
    # This will print the content of
    # the response.

    print(response.text)
else:
    print("Failed to retrieve data. Status code:", response.status_code)
```

Output:

```
<!doctype html><htmllang="en"><head><meta http-equiv="Content-
Type" content="text/html; charset=utf-8"/><meta name="viewport"
content="width=device-width, initial-scale=1.0"/><link
rel="stylesheet" href="/public/css/styles.css"/><title> Springer -
International Publisher Science, Technology, Medicine

...
```

Two versions of a code are shown here. One that requests a non-discriminating server response, and the other version that requests a response based on different parameters (typically used for APIs). In the first version, the Python script uses the *requests* library to fetch HTML content from a website. Initially, the script imports the *requests* module, which is a popular HTTP library used for sending HTTP requests in Python. The script then defines a URL variable *url* which stores the address of the website to be accessed. In this example, "http://www.springer.com" is used as a placeholder. Following this, the script makes a *GET* request to the specified URL using *requests.get*(*url*). This function sends a *GET* request to the server that hosts the website and returns a response object. The script then checks if the request was successful by examining the *status_code attribute* of the response object. If the server responded with a status code of 200, it indicates that

the request was successful, and the website content was retrieved without any issues. In this case, the script prints the HTML content of the website, accessed via *response.text*. If the status code is not 200, it suggests that there was an issue with fetching the webpage, such as the website being unreachable or the URL being incorrect. In this situation, the script outputs "Failed to retrieve the webpage" to indicate that it could not successfully fetch the HTML content. Thus, this first script is a basic example of web scraping or fetching website data. In the second version, the provided Python script demonstrates how to send a GET request with query parameters to a server using the same *requests* library. As before, the script imports the *requests* library, which is essential for making HTTP requests. The *url* variable is then defined with the desired endpoint, in this case, "https://springer.com/api". To include query parameters in the GET request, a dictionary named *params* is created. This dictionary contains key-value pairs corresponding to the parameters and their values that must be send to the server. For example, *params* = {'param1': 'value1', 'param2': 'value2'} defines two parameters: "param1" with the value "value1", and "param2" with the value "value2". The actual GET request is made with the line response = *requests.get*(*url*, *params* = *params*). Here, the *requests.get* function is called with the URL and the params dictionary. Next, the *requests* library takes care of properly encoding the parameters and appending them to the URL. After sending the request, the script checks if it was successful by examining *response.status_code*. A status code of 200 indicates success. If the request is successful, *response.text* is printed, which contains the response data from the server. In case of a failure, indicated by any status code other than 200, the script prints a failure message along with the received status code. This approach is particularly useful for interacting with APIs or web services where specific data needs to be retrieved based on given parameters, ensuring that the request is both flexible and dynamically constructed based on the parameters defined in the params dictionary.

References

1. P.A. Gagniuc, An Introduction to Programming Languages: Simultaneous Learning in Multiple Coding Environments. Synthesis Lectures on Computer Science (Springer Nature, 2023), pp. 1–280.
2. A. Kumar, P.S. Panda, A survey: how python pitches in IT-world, in *International Conference on Machine Learning, Big Data, Cloud and Parallel Computing (COMITCon)*, Faridabad, India (2019), pp. 248–251
3. K. David, S. Paul, M. Ivan, E. Tomz, *Classic Computer Science Problems in Python* (Manning, 2019)
4. A. Nagpal, G. Gabrani, Python for data analytics, scientific and technical applications, in *Amity International Conference on Artificial Intelligence (AICAI)*, Dubai, United Arab Emirates (2019), pp. 140–145
5. J. Guniš, L. Šnajder, Z. Tkáčová, V. Gunišová, Inquiry-based Python programming at secondary schools, in *43rd International Convention on Information, Communication and Electronic Technology (MIPRO)*, Opatija, Croatia (2020), pp. 750–754
6. T.R. Fernandes, L.R. Fernandes, T.R. Ricciardi, L.F. Ugarte, Python programming language for power system analysis education and research, in *IEEE PES Transmission & Distribution Conference and Exhibition*, Lima, Peru (2018), pp. 1–5
7. A.P. Lorandi Medina, G.M. Ortigoza Capetillo, G.H. Saba, M.A.H. Pérez, P.J. García Ramírez, A simple way to bring Python to the classrooms, in *IEEE International Conference on Engineering Veracruz (ICEV)*, Boca del Rio, Mexico (2020), pp. 1–6
8. X. Liu, H. Xu, School-enterprise cooperation on Python data analysis teaching, in *14th International Conference on Computer Science & Education (ICCSE)*, Toronto, ON, Canada (2019), pp. 278–281
9. C. Francois, *Deep Learning with Python*, 2nd edn. (Manning, 2021)
10. C.D. López, M. Cvetković, P. Palensky, Enhancing power factory dynamic models with Python for rapid prototyping, in *IEEE 28th International Symposium on Industrial Electronics (ISIE)*, Vancouver, BC, Canada (2019), pp. 93–99
11. I. Stančin, A. Jović, An overview and comparison of free Python libraries for data mining and big data analysis, in *42nd International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*, Opatija, Croatia (2019), pp. 977–982
12. I. Grout, W.A.P. de Ferreira, A.C.R. da Silva, On-line electrical supply generation fuel mix data analysis using Python and TensorFlow, in *International Conference on Power, Energy and Innovations (ICPEI)*, Pattaya, Thailand (2019), pp. 20–23

P. A. Gagniuc, *Coding Examples from Simple to Complex*, Synthesis Lectures on Computer Science, https://doi.org/10.1007/978-3-031-53812-4

13. R. Smith, Performance of MPI codes written in Python with NumPy and mpi4py, in *6th Workshop on Python for High-Performance and Scientific Computing (PyHPC)*, Salt Lake City, UT, USA (2016), pp. 45–51
14. M. Shah, R. Shenoy, R. Shankarmani, Natural language to Python source code using transformers, in *International Conference on Intelligent Technologies (CONIT)*, Hubli, India (2021), pp. 1–4
15. T. Silas, P. Bill, T. Christopher, R. René, *Python for ArcGIS Pro: Automate Cartography and Data Analysis Using ArcPy, ArcGIS API for Python, Notebooks, and Pandas* (Packt Publishing, 2022)
16. E. Chou, *Mastering Python Networking: Utilize Python Packages and Frameworks for Network Automation, Monitoring, Cloud, and Management* (Packt Publishing, 2023)
17. S.F. Lott, P. Dusty, *Python Object-Oriented Programming: Build Robust and Maintainable Object-Oriented Python Applications and Libraries* (Packt Publishing, 2021)
18. P. Gagniuc, C. Ionescu-Tîrgovişte, Dynamic block allocation for biological sequences. Proc. Rom. Acad. Ser. B **15**(3), 233–240 (2013)
19. N. Avinash, F. Armando, I. Ivan, *Python Data Analysis: Perform Data Collection, Data Processing, Wrangling, Visualization, and Model Building Using Python* (Packt Publishing, 2021)
20. P. Gagniuc, C. Ionescu-Tîrgovişte, C.H. Rădulescu, Automatic growth detection of cell cultures through outlier techniques. Int. J. Comput. Commun. **8**(3), 407–415 (2013)
21. K. Thomas, P. Gagniuc, E. Gagniuc, Moonlighting genes harbor antisense ORFs that encode potential membrane proteins. Sci. Rep. **13**, 12591 (2023)
22. P. Gagniuc, et al., A sensitive method for detecting dinucleotide islands and clusters through depth analysis. Romanian J. Diabet. Nutrit. Metab. Dis. **18**(2), 165–170 (2011)
23. J. Singh, J. Singh, G. Singh, N. Kaur, Exploratory data analysis for interpreting model prediction using Python, in *International Conference on Smart Generation Computing, Communication and Networking (SMART GENCON)*, Bangalore, India (2022), pp. 1–6
24. P.A. Gagniuc, C. Ionescu-Tirgoviste, E. Gagniuc, M. Militaru, L.C. Nwabudike, B.I. Pavaloiu, A. Vasilăţeanu, N. Goga, G. Drăgoi, I. Popescu, S. Dima, Spectral forecast: a general purpose prediction model as an alternative to classical neural networks. Chaos **30**, 033119–033126 (2020)
25. P.A. Gagniuc, *Algorithms in Bioinformatics: Theory and Implementation* (Wiley, USA, Hoboken, New Jersey, 2021)
26. P.A. Gagniuc, *Markov Chains: From Theory to Implementation and Experimentation* (Wiley, Hoboken, NJ, USA, 2017)
27. F. Rigueira, J. Bernardino, I. Pedrosa, Extraction of information from log files using Python programming and Tableau, in *15th Iberian Conference on Information Systems and Technologies (CISTI)*, Seville, Spain (2020), pp. 1–7