Programming in Python

Module 2

Rijin IK

Assistant Professor
Department of Computer Science and Engineering
Vimal Jyothi Engineering College
Chemperi

February 28, 2024

Outline

Accessing Characters and Substrings in Strings

The Structure of Strings

- A string is a data structure in Python.
- Unlike an integer, a string is a compound unit consisting of several pieces of data.
- A string is a sequence of zero or more characters.
- In Python, strings can be represented using either single or double quote marks.

```
>>> "Hi there!"
'Hi there!'
>>> ""
''
>>> 'R'
'R'
```

Accessing Characters and Substrings in Strings

Length of a String

- The length of a string is the number of characters it contains.
- Python's len function returns the length of a string when passed as an argument.

```
>>> len("Hi there!")
9
>>> len("")
0
```

Positions of Characters

- The positions of a string's characters are numbered from 0 (left) to length - 1 (right).

```
# Positions:   0 1 2 3 4 5 6 7 8
# String:      H i   t h e r e !
```

Accessing Characters and Substrings in Strings

Immutability of Strings

- A string is an immutable data structure.
- This means that its internal data elements, the characters, can be accessed but cannot be replaced, inserted, or removed.

```
# Example of immutability
string_example = "Python"

# This will result in an error
string_example[0] = 'J'

ERROR!
Traceback (most recent call last):
File "<string>", line 4, in <module>
TypeError: 'str' object does not support item assignment
```

Accessing Characters and Substrings in Strings

The Subscript Operator

- The subscript operator ([]) allows for accessing individual characters in a string.
- The basic form is

```
<a string>[<an integer expression>].
```

- The string is the part you want to inspect, and the integer expression is the index or position of the character.
- Examples using the string "Alan Turing" show the use of the subscript operator to access specific characters.
- Typically, positions range from 0 to the length of the string minus 1.
- Python allows the use of negative subscript values to access characters near the end of a string, counting backward from -1.

Accessing Characters and Substrings in Strings

The Subscript Operator

```
>>> name = "Alan Turing"
>>> name[0] # Examine the first character
'A'
>>> name[3] # Examine the fourth character
'n'
>>> name[len(name)] # Oops! An index error!
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
IndexError: string index out of range
>>> name[len(name) - 1] # Examine the last character
'g'
>>> name[-1] # Shorthand for the last character
'g'
>>> name[-2] # Shorthand for next to last character
'n'
```

Accessing Characters and Substrings in Strings

Slicing for Substrings in Python:

- Slicing in Python refers to the process of extracting a portion (substring) from a string using the subscript operator with a colon (:).
- This powerful technique allows programmers to efficiently manipulate and retrieve specific sections of a string.

```
>>> name = "myfile.txt" # The entire string
>>> name[0:]
'myfile.txt'
>>> name[0:1] # The first character
'm'
>>> name[0:2] # The first two characters
'my'
>>> name[:len(name)] # The entire string
'myfile.txt'
>>> name[-3:] # The last three characters
'txt'
>>> name[2:6] # Drill to extract 'file'
'file'
```

Data encryption

Data encryption:
Caesar Cipher Encryption and Decryption in Python:

- The provided Python scripts (encrypt.py and decrypt.py) implement a simple encryption and decryption technique known as the Caesar cipher.
- The Caesar cipher is a substitution cipher where each character in the plaintext is shifted by a fixed number of positions down the alphabet.
- Encryption Script (encrypt.py):

Data encryption

Data encryption:

Caesar Cipher Encryption and Decryption in Python:

- Encryption Script (encrypt.py):

```
# encrypt.py
plainText = input("Enter a one-word, lowercase message: ")
distance = int(input("Enter the distance value: "))
code = ""

for ch in plainText:
    ordvalue = ord(ch)
    cipherValue = ordvalue + distance

    if cipherValue > ord('z'):
        cipherValue = ord('a') + (cipherValue - ord('z') - 1)

    code += chr(cipherValue)

print("Encrypted Message:", code)
```

Data encryption

- Decryption Script (decrypt.py):

```
# decrypt.py
cipherText = input("Enter the encrypted message: ")
distance = int(input("Enter the distance value: "))
decodedText = ""

for ch in cipherText:
    ordvalue = ord(ch)
    decryptedValue = ordvalue - distance

    if decryptedValue < ord('a'):
        decryptedValue = ord('z') - (ord('a') - decryptedValue - 1)

    decodedText += chr(decryptedValue)

print("Decrypted Message:", decodedText)
```

Data encryption

- The encrypt.py script takes a one-word lowercase message and a distance value as input, then performs Caesar cipher encryption.
- The decrypt.py script takes the coded text and the same distance value as input, then performs Caesar cipher decryption.
- The examples show the encryption of "invaders" with a distance of 3, resulting in "Lqydghuv," and the decryption of "Lqydghuv" with the same distance, yielding the original message "invaders."
- The scripts can be extended to handle spaces and punctuation marks for more comprehensive use.

Strings and Number System

Strings and Number Systems

- Decimal number system (base 10 number sys)
    - 0 to 9
- Binary number system (base 2)
    - 0 and 1
- Octal number system (base 8)
    - 0 to 7
- Hexadecimal number system (base 16)
    - 0 to 9, A, B, C, D, E, F

Strings and Number System

Number system examples

```
415 in binary notation 1100111112
415 in octal notation 6378
415 in decimal notation 41510
415 in hexadecimal notation 19F16
```

Strings and Number System

Converting Binary to Decimal

Binary: 1100111_2

Position: $2^6 \quad 2^5 \quad 2^4 \quad 2^3 \quad 2^2 \quad 2^1 \quad 2^0$

$$\text{Decimal} = (1 \times 2^6) + (1 \times 2^5) + (0 \times 2^4) + (0 \times 2^3)$$
$$+ (1 \times 2^2) + (1 \times 2^1) + (1 \times 2^0)$$
$$= 64 + 32 + 0 + 0 + 4 + 2 + 1$$
$$= 103$$

Strings and Number System

```
"""
File: binarytodecimal.py
Converts a string of bits to a decimal integer.
"""
bitString = input("Enter a string of bits: ")
decimal = 0
exponent = len(bitString) - 1
for digit in bitString:
    decimal = decimal + int(digit) * 2 ** exponent
    exponent = exponent - 1
print("The integer value is", decimal)



"""
output
Enter a string of bits: 1111
The integer value is 15
Enter a string of bits: 101
The integer value is 5
"""
```

Strings and Number System

Converting Decimal to Binary

```
"""
Converts a decimal integer to a string of bits.
"""
decimal = int(input("Enter a decimal integer: "))
if decimal == 0:
    print(0)
else:
    bitString = ""
    while decimal > 0:
        remainder = decimal % 2
        decimal = decimal // 2
        bitString = str(remainder) + bitString
print("The binary representation is", bitString)
'''
output
Enter a decimal integer: 5
The binary representation is 101
'''
```

Strings and Number System

Exercise

- Write a Python script to convert the following
  1. Octal to Binary
  2. Binary to Octal
  3. Hexadecimal to Binary
  4. Binary to Hexadecimal

Strings and Number System

Exercise

- Write a Python script to convert the following
  1. Octal to Binary
  2. Binary to Octal
  3. Hexadecimal to Binary
  4. Binary to Hexadecimal

String methods

String methods

- In Python, strings are objects, and there are various built-in methods (or functions) that you can use to manipulate and operate on strings.

String Method 1: s.center(width)

s.center(width)

Returns a copy of s centered within the given number of columns.

```
s = "example"
result = s.center(15)
print(result)
"""
Output:
    example
"""
```

String Method 2: s.count(sub [, start [, end]])

s.count(sub [, start [, end]])
Returns the number of non-overlapping occurrences of substring sub in s. Optional arguments start and end are interpreted as in slice notation.

```
s = "example example"
count = s.count("example")
print(count)
```

Output:

```
2
```

String Method 3: s.endswith(sub)

s.endswith(sub)

Returns True if s ends with sub or False otherwise.

```python
s = "example.txt"
result = s.endswith(".txt")
print(result)
```

Output:

```
True
```

String Method 4: `s.find(sub [, start [, end]])`

s.find(sub [, start [, end]])
Returns the lowest index in `s` where substring `sub` is found. Optional arguments `start` and `end` are interpreted as in slice notation.

```
s = "example"
index = s.find("am")
print(index)
```

Output:

```
2
```

String Method 5: s.isalpha()

s.isalpha():

Returns True if s contains only letters or False otherwise.

```
s = "abc"
result = s.isalpha()
print(result)
```

Output:

```
True
```

String Method 6: `s.isdigit()`

s.isdigit():

Returns `True` if `s` contains only digits or `False` otherwise.

```
s = "123"
result = s.isdigit()
print(result)
```

Output:

```
True
```

String Method 7: s.join(sequence)

s.join(sequence):
Returns a string that is the concatenation of the strings in the sequence. The separator between elements is s.

```
s = "-"
sequence = ["a", "b", "c"]
result = s.join(sequence)
print(result)
```

Output:

```
a-b-c
```

String Method 8: `s.lower()` & `s.upper()`

s.lower():
Returns a copy of s converted to lowercase.

```
s = "Hello World"
result = s.lower()
print(result)
```

Output:

```
hello world
```

s.upper():
To convert the string to uppercase, you can use the upper() method

```
s = "Hello World"
result = s.upper()
print(result)
```

Output:

```
HELLO WORLD
```

String Method 9: `s.replace(old, new [, count])`

s.replace(old, new [, count]):
Returns a copy of `s` with all occurrences of substring `old` replaced by `new`. If the optional argument `count` is given, only the first `count` occurrences are replaced.

```
s = "apple apple orange apple"
result = s.replace("apple", "banana", 2)
print(result)
```

Output:

```
banana banana orange apple
```

String Method 10: s.split([sep])

s.split([sep]):
Returns a list of the words in s, using sep as the delimiter string. If sep is not specified, any whitespace string is a separator.

```
s = "apple orange banana"
result = s.split()
print(result)
```

Output:

```
['apple', 'orange', 'banana']
```

```
s = "apple orange banana"
result = s.split('a')
print(result)
```

Output:

```
['', 'pple or', 'nge b', 'n', 'n', '']
```

String Method 11: s.startswith(sub)

s.startswith(sub):

Returns True if s starts with sub or False otherwise.

```
s = "Hello World"
result = s.startswith("Hello")
print(result)
```

Output:

```
True
```

String Method 12: s.strip()

s.strip([aString]):
Returns a copy of s with leading and trailing whitespace (tabs, spaces, newlines) removed.

```
s = "   Hello World   "
result = s.strip()
print(result)
```

Output:

```
Hello World
```

Writing Text to a File - Opening a File for Output

Opening a File for Output:

To output data to a text file, use Python's `open` function with the mode string `'w'`.

```
f = open("myfile.txt", 'w')
```

This opens a connection to a file named `myfile.txt` for output.

Writing Text to a File - Writing String Data

Writing String Data

Use the `write` method with the file object to write string data.

```
f.write("First line.\nSecond line.\n")
```

This writes the specified string to the file, including newline characters to create separate lines.

Writing Text to a File - Closing the File

Closing the File

After writing, close the file using the `close` method.

```
f.close()
```

This ensures that the file is properly closed after writing.

Writing Numbers to a File

Writing Numbers to a File

The `write` method expects a string as an argument. Therefore, other types of data, such as integers, must be converted to strings before being written to an output file.

Example: Writing Random Integers

```python
import random
f = open("integers.txt", 'w')
for count in range(500):
    number = random.randint(1, 500)
    f.write(str(number) + '\n')
f.close()
```

This code generates 500 random integers between 1 and 500 and writes them to a text file named `integers.txt` with newline characters as separators.

Reading Text from a File - Entire Contents

Reading Text from a File - Entire Contents

To read the entire contents of a file as a single string, use the `read` method.

```
f = open("myfile.txt", 'r')
text = f.read()
print(text)
```

This reads and prints the entire contents of the file, including newline characters.

Reading Text from a File - Line by Line

Reading Text from a File - Line by Line

To read and process text one line at a time, use a for loop.

```
f = open("myfile.txt", 'r')
for line in f:
    print(line)
```

This loop views the file object as a sequence of lines, printing each line.

Reading Text from a File - Specified Number of Lines

Reading Text from a File - Specified Number of Lines

To read a specified number of lines from a file, use the `readline` method.

```
f = open("myfile.txt", 'r')
while True:
    line = f.readline()
    if line == "":
        break
    print(line)
```

This loop reads and prints all lines using `readline`.

Reading Numbers from a File - Line by Line

Reading Numbers from a File - Line by Line

To read numbers line by line from a file, use a for loop to iterate through the lines.

```
f = open("integers.txt", 'r')
theSum = 0
for line in f:
    line = line.strip()
    number = int(line)
    theSum += number
print("The sum is", theSum)
```

This example reads integers separated by newlines and prints their sum.

Reading Numbers from a File - Handling Spaces

Reading Numbers from a File - Handling Spaces

To handle numbers separated by spaces, use the `split` method and process each word in a nested loop.

```
f = open("integers.txt", 'r')
theSum = 0
for line in f:
    wordlist = line.split()
    for word in wordlist:
        number = int(word)
        theSum += number
print("The sum is", theSum)
```

This example reads integers separated by spaces or newlines and prints their sum.

Summary - File Operations

Operation	Description
`open("filename", 'r')`	Open file for reading
`open("filename", 'w')`	Open file for writing
`read()`	Read entire contents of the file
`readline()`	Read a line from the file
`strip()`	Remove leading and trailing whitespace
`split()`	Split a string into a list of words

Table: File Operations

Case Study: Text Analysis

- In 1949, Dr. Rudolf Flesch introduced the Flesch Index for text readability.
- The index is based on syllables per word and words per sentence.
- Scores range from 0 to 100, indicating readability for different grade levels.

Request:

- Write a program to compute the Flesch Index for a given text file.

Analysis:

- Definitions: Word, Sentence, Syllable (see Definitions of items used in the text analysis program).
- Flesch's formula for Index (F) and Grade Level Equivalent (G).

Analysis

Term	**Definition**
Word	Any sequence of non-whitespace characters.
Sentence	Any sequence of words ending in a period, question mark, exclamation point, colon, or semicolon.
Syllable	Any word of three characters or less; or any vowel (a, e, i, o, u) or pair of consecutive vowels, except for a final -es, -ed, or -e that is not -le.

Table: Definitions of items used in the text analysis program

Analysis

Flesch's Formula

The Flesch's Formula to calculate the index F is given by:

$$F = 206.835 - 1.015\left(\frac{\text{words}}{\text{sentences}}\right) - 84.6\left(\frac{\text{syllables}}{\text{words}}\right)$$

Analysis

Flesch-Kincaid Grade Level Formula

The Flesch-Kincaid Grade Level Formula for computing the Equivalent Grade Level G is given by:

$$G = 0.39\left(\frac{\text{words}}{\text{sentences}}\right) + 11.8\left(\frac{\text{syllables}}{\text{words}}\right) - 15.59$$

Design

Program Tasks:

1. Receive the filename from the user.
2. Count sentences, words, and syllables.
3. Compute Flesch Index.
4. Compute Grade Level Equivalent.
5. Print results.

Implementation:

- Open the file, read text, and count sentences/words/syllables.
- Use formulas to compute Flesch Index and Grade Level.
- Output the results.

Implementation

```
"""
Program: textanalysis.py
Author: Ken
Computes and displays the Flesch Index and the Grade
Level Equivalent for the readability of a text file.
"""
# Take the inputs
fileName = input("Enter the file name: ")
inputFile = open(fileName, 'r')
text = inputFile.read()
```

Implementation

```python
# Count the sentences
sentences = text.count('.') + text.count('?') + \
    text.count(':') + text.count(';') + \
    text.count('!')
# Count the words
words = len(text.split())
# Count the syllables
syllables = 0
vowels = "aeiouAEIOU"
for word in text.split():
    for vowel in vowels:
        syllables += word.count(vowel)
    for ending in ['es', 'ed', 'e']:
        if word.endswith(ending):
            syllables -= 1
    if word.endswith('le'):
        syllables += 1
```

Implementation

```python
# Compute the Flesch Index and Grade Level
index = 206.835 - 1.015 * (words / sentences) - \
        84.6 * (syllables / words)
level = round(0.39 * (words / sentences) + 11.8 * \
        (syllables / words) - 15.59)
# Output the results
print("The Flesch Index is", index)
print("The Grade Level Equivalent is", level)
print(sentences, "sentences")
print(words, "words")
print(syllables, "syllables")
```

Testing

Bottom-Up Testing:

- Test individual tasks independently.
- Use a driver script for Flesch Index and Grade Level computation.

Integration Testing:

- Combine all tasks into the complete program.
- Test with short and long files.
- Compare results with expected outcomes.

Design with Functions

Design with Functions

- A function in Python is a block of code that performs a specific task.
- Functions are used to break down a program into smaller, modular chunks.
- They enhance code organization, manageability, and reusability.
- Syntax of a Function:

```
def function_name(parameters):
    # statements
    return [expression]
```

  - def: Keyword marking the start of the function header.
  - function_name: Unique identifier for the function.
  - parameters: Values passed to the function (optional).
  - statements: Python statements forming the function body (indented).
  - return: Optional statement to return a value from the function.

Design with Functions

Calling a Function

- To Call a Function:

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

my_function()  # Calling the function
```

The Return Statement

- Used to exit a function and return a value.

```
Syntax: return [expression]
```

- If no expression or return is present, the function returns None.

Design with Functions

Types of Functions

- Built-in Functions:
  - Provided by Python, e.g., dir(), len(), abs().
- User-Defined Functions:
  - Created by users for specific tasks.

Advantages of Functions

Advantages of Functions

- Reduction of Code Duplication:
  - Functions reduce redundancy by allowing code to be reused.

```
def summation(lower, upper):
    """Arguments: A lower bound and an upper bound
    Returns: the sum of the numbers from lower through upper
    """
    result = 0
    while lower <= upper:
        result += lower
        lower += 1
    return result

>>> summation(1,4) # The summation of the numbers 1..4
10
>>> summation(50,100) # The summation of the numbers 50..100
3825
```

  - Without this function, the summation algorithm would need to be repeated.

Advantages of Functions

- Breaking Down Complex Problems:
    - Functions help break large, complex problems into smaller, manageable parts.
    - Improve code clarity and maintainability.

Functions as Abstraction Mechanism

Functions as Abstraction Mechanism

- Abstraction:
  - Mechanism to simplify or hide complexity.
  - Functions serve as abstraction mechanisms in a program.
- Elimination of Redundancy:
  - Functions eliminate redundant code by centralizing algorithms in one place.
  - Debugging and maintenance become easier.
- Hiding Complicated Details:
  - Functions hide complex details, allowing users to interact with a high-level interface.
  - Users only need to understand the function's purpose, not its internal implementation.
- Functions Support General Methods with Systematic Variations:
  - Functions in Python act as versatile tools that can apply a general method to solve various problems.
  - By allowing systematic variations through parameters, functions provide adaptability for handling different scenarios within a specific class of problems.

Functions as Abstraction Mechanism

- Functions Support the Division of Labor
  - functions in a well-designed computer program serve as specialized agents, enforcing a division of labor by assigning specific tasks to each function.
  - This organization enhances clarity, maintainability, and the overall efficiency of the system, ensuring that each function contributes coherently to the program's common goal.

Problem solving with top-down design

Top-Down Design

- Top-down design simplifies complex problems:
  - Break down a big problem into smaller, more manageable parts.
- Use functions for isolated components:
  - Assign solutions for each smaller part to specific functions.
- Decomposition can go deeper:
  - Subproblems may themselves have further breakdowns into smaller problems.
- Stepwise refinement builds the solution:
  - Create functions for each piece of the problem, slowly building up the complete solution in a detailed way.
- **Advantages:** Enhances code organization, readability, and modularity.

Problem solving with top-down design

The Design of the Text-Analysis Program

- The design of the Text-Analysis program involves structuring the processing tasks into programmer-defined functions.
- While simple input and output tasks can be handled within a main function, more complex processes, such as counting sentences, words, and syllables, are decomposed into smaller functions.
- The structure chart visually represents the relationships between these functions, demonstrating how data flows between them.
- The main function initiates the design, and decomposition leads to lower-level functions, each serving a specific computational task.
- This structured design facilitates efficient development by focusing on the results needed by the main function from its collaborating functions.

Problem solving with top-down design

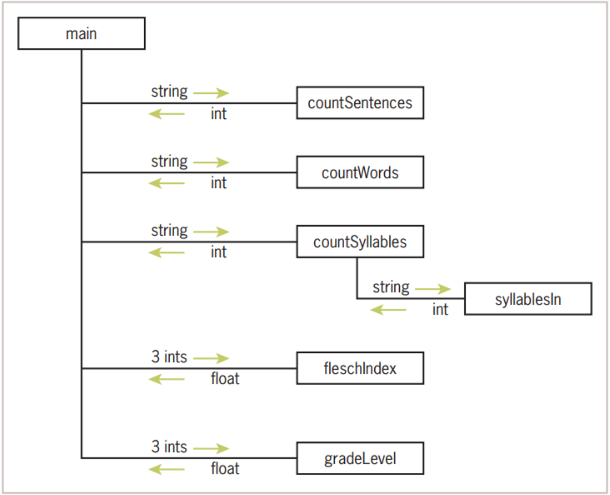


Figure: A structure chart for the text-analysis program

Problem solving with top-down design

The Design of the Sentence-Generator Program

- The Sentence-Generator program's design centers on a main loop for generating sentences based on user input.
- The main function handles simple I/O and loop logic.
- The sentence generation is achieved by decomposing the problem according to grammar rules, where each rule corresponds to a function addressing a specific linguistic structure.
- This top-down design reflects the grammar's hierarchical structure, illustrating how the program's structure mirrors the problem's organization.
- In contrast to the text analyzer, where functions receive data via parameters, the sentence generator functions draw from a common pool of data defined at the module's start.
- This pool could have been passed as arguments to functions, but using a shared data pool simplifies the design and improves program maintainability by avoiding the need to pass unused arguments to certain functions, streamlining the overall structure.

Problem solving with top-down design

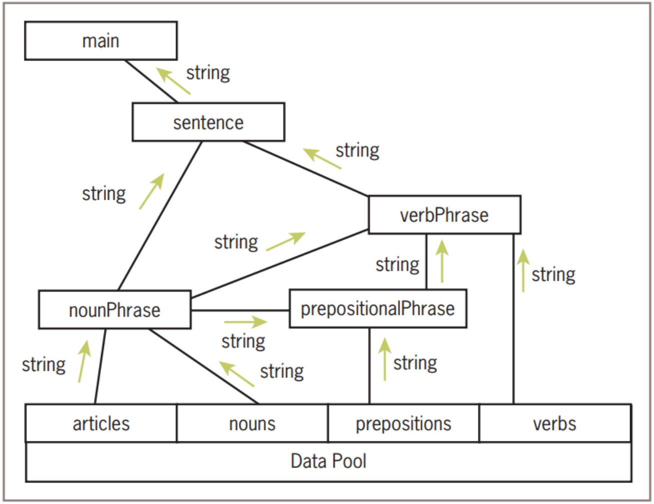


Figure: A structure chart for the sentence-generator program

Problem solving with top-down design

The Design of the Doctor Program

- At the top level, the designs of the doctor program and the sentence-generator program share similarities, both featuring main loops for user interaction.
- In the case of the doctor program, the structure chart reveals a separation of responsibilities between the main and reply functions.
- This design approach, known as responsibility-driven design, allocates distinct roles: main handles user interaction, while reply implements the ”doctor logic” of generating appropriate replies.
- The reply function is tasked with choosing a response strategy, seeking assistance from other functions like changePerson to execute each option.
- This separation enhances maintainability, allowing easy addition or modification of response strategies without altering the core logic.
- The data flow in the doctor program combines elements from the text analyzer and sentence generator, with functions receiving data from both user input and a shared data pool.

Problem solving with top-down design

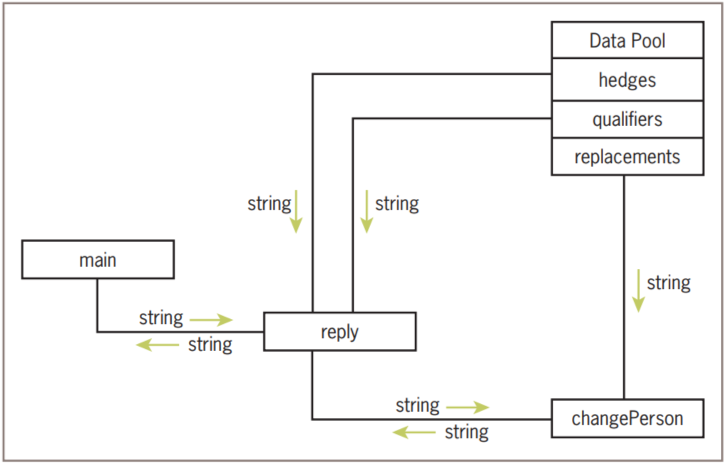


Figure: A structure chart for the doctor program

Design with recursive functions

Introduction to Recursive Functions

- Recursive design decomposes a complex problem into smaller problems of the same form.
- Recursive functions call themselves and must have a base case to prevent infinite recursion.
- **Base case:** A condition that, when met, stops the recursive process.

Design with recursive functions

Defining a Recursive Function:

- Syntax: def function_name(parameters):
- **Example:** Convert an iterative algorithm to a recursive function for displaying a range of numbers.

```
def displayRange(lower, upper):
    if lower <= upper:
        print(lower)
        displayRange(lower + 1, upper)
```

Design with recursive functions

Tracing a Recursive Function:

- Tracing recursive calls helps understand how the function works.
- **Example:** Here's an example of a factorial function and its trace

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

# Trace the factorial function with an example call
result = factorial(4)
print("Result:", result)
```

Design with recursive functions

Tracing a Recursive Function Cont..

```
Initial Call: factorial(4)
    Checks if n is 0 or 1 (false in this case).
    Calculates result = 4 * factorial(3).
Recursive Call: factorial(3)
    Checks if n is 0 or 1 (false in this case).
    Calculates result = 3 * factorial(2).
Recursive Call: factorial(2)
    Checks if n is 0 or 1 (false in this case).
    Calculates result = 2 * factorial(1).
Recursive Call: factorial(1)
    Checks if n is 0 or 1 (true).
    Returns 1.
    Unwinding recursion: result = 2 * 1.
Unwinding Recursion: Back to factorial(2)
    result = 3 * 2.
Unwinding Recursion: Back to factorial(3)
    result = 4 * 6.
Final Result: Result: 24
```

Design with recursive functions

Recursive Definitions and Fibonacci Sequence:

- Recursive definitions express values in terms of base cases and recursive cases.
- **Example:** Recursive definition of Fibonacci number.

```python
def fibonacci_recursive(a, b, i, n):
    if i <= n:
        print(a)
        c = a + b
        fibonacci_recursive(b, c, i + 1, n)

# Input: Number of terms in the Fibonacci sequence
n = int(input("Enter the number of terms: "))

# Initialize the sequence with the first two terms (0 and 1)
a, b = 0, 1

# Start the recursive function
fibonacci_recursive(a, b, 1, n)
```

Design with recursive functions

Infinite Recursion:

- Infinite recursion(continue executing forever) occurs when a function lacks a proper base case.
- **Example:** A function leading to infinite recursion.

```
def runForever(n):
    if n > 0:
        runForever(n)
    else:
        runForever(n - 1)
```

Design with recursive functions

Costs and Benefits of Recursion:

- Recursive solutions are elegant but may incur overhead in terms of memory and processing time.
- Infinite recursion can lead to stack overflow errors.
- A balance is needed between elegance and performance in choosing recursive vs. iterative solutions.

Managing a program's namespace

Managing a program's namespace:
Module Variables, Parameters, and Temporary Variables:

- **Module variables:** Introduced at the module level, visible throughout the module.
- **Parameters:** Introduced in function headers, visible within the function, invisible outside.
- **Temporary variables:** Introduced in function bodies, visible only within the function.

Managing a program's namespace

Scope:

- Scope defines the area of program text in which a variable's name refers to a given value.
- Temporary variables and parameters have local scope, limited to their function.
- Module variables have a broader scope, covering the entire module.

Managing a program's namespace

- In this example global_variable has module-level scope, accessible inside the function and local_variable has local scope, visible only within the example_function.

```
# Module-level variable
global_variable = "I am a global variable"

def example_function():
    # Local variable within the function
    local_variable = "I am a local variable"
    print(local_variable)
    print(global_variable)  # Accessing the global variable

# Try to access the local_variable outside the function - it will
    result in an error
# print(local_variable)  # Uncommenting this line will result in an
     error

# Calling the function
example_function()
```

Managing a program’s namespace

Lifetime:

- A variable’s lifetime is the period during program execution when it has associated memory storage.
- Module variables generally exist for the entire program’s lifetime.
- Temporary variables and parameters exist during function calls and are reclaimed afterward.

Managing a program's namespace

- In this example:
  temporary_variable has a lifetime limited to the function call. Once the function execution completes, the memory associated with temporary_variable is reclaimed.

```python
def function_with_temporary_variable():
    temporary_variable = "I exist temporarily"
    print(temporary_variable)

# Calling the function
function_with_temporary_variable()

# Trying to access the temporary_variable outside the function - it
     will result in an error
# print(temporary_variable)  # Uncommenting this line will result
    in an error
```

Managing a program's namespace

Using Keywords for Default and Optional Arguments:

- Functions often require specific arguments but may have optional ones with default values. Syntax:

```
def function_name(required, key1=default1, key2=default2):
```

Example: The range function and its optional arguments.

```
def example(required, option1=2, option2=3):
    print(required, option1, option2)

# Different ways to call the function
example(1)  # Output: 1 2 3 (Use all defaults)
example(1, 10)  # Output: 1 10 3 (Override the first default)
example(1, 10, 20)  # Output: 1 10 20 (Override all defaults)
example(1, option2=20)  # Output: 1 2 20 (Override the second
    default)
example(1, option2=20, option1=10)  # Output: 1 10 20(In any order)
```

Managing a program's namespace

Benefits of Default Arguments:

- Default arguments make a function more flexible without making it complicated.
- Programmers can easily use functions in typical situations and modify them as necessary.
- Python's built-in functions often use default arguments to provide versatility.

Higher-Order Functions

Higher-Order Functions:

- Higher-order functions are functions that either take other functions as arguments or return functions as results.

Functions as First-Class Data Objects

- Python treats functions as first-class citizens, meaning they can be treated like any other data type.
- They can be assigned to variables, passed as arguments to other functions, returned as values from other functions, and stored in data structures like lists and dictionaries.

Higher-Order Functions

```
# Functions as first-class objects
def square(x):
    return x ** 2

def cube(x):
    return x ** 3

# Assigning functions to variables
power_function = square
print(power_function(4))  # Output: 16

# Storing functions in a list
function_list = [square, cube]
print(function_list[1](3))  # Output: 27
```

Higher-Order Functions

Mapping

- Mapping in Python is a higher-order function that utilizes the **map** function, which takes two arguments: a function and an iterable (sequence).
- The function is applied to each element of the iterable, producing a new sequence as the output.

```
words = ["231", "20", "-45", "99"]  # Initial list of strings
# Apply the int function to each element using map
# This does not modify the original list but returns an iterator
mapped_result = map(int, words)
# Display the original 'words' list
print(words)  # Output: ['231', '20', '-45', '99']

# Reset the 'words' variable by converting the iterator to a list
# Now the 'words_new' list contains integers
words_new = list(mapped_result)
# Display the updated 'words_new' list
print(words_new)  # Output: [231, 20, -45, 99]
```

Higher-Order Functions

Filtering

- Filtering is a higher-order function that applies a given predicate (a function that returns a boolean) to each element of a sequence, retaining only the elements that satisfy the condition.
- Python includes a **filter** function for this purpose.

```
# Define a function 'odd' that checks if a number is odd
def odd(n):
    return n % 2 == 1

# Use the filter function to apply the 'odd' function to each
    element in the range(10)
# The filter function returns an iterator containing elements for
    which the 'odd' function is True
result = list(filter(odd, range(10)))

# Print the result, which is a list of odd numbers from the range
    (10)
print(result)  # Output: [1, 3, 5, 7, 9]
```

Higher-Order Functions

Reducing

- Reducing is a higher-order function that repeatedly applies a function of two arguments cumulatively to the items of a sequence, reducing it to a single value.
- Python includes a **reduce** function for this purpose.

```
from functools import reduce
def add(x, y):
    return x + y

def multiply(x, y):
    return x * y
data = [1, 2, 3, 4]

sum_result = reduce(add, data)
print(sum_result)  # Output: 10

product_result = reduce(multiply, data)
print(product_result)  # Output: 24
```

Higher-Order Functions

Lambda Expressions

- Lambda expressions allow the creation of anonymous functions, which are useful for short-term use in higher-order functions without the need for a formal function definition.
- The syntax of a lambda is

```
lambda <argname-1, ..., argname-n>: <expression>
```

- When the lambda is applied to its arguments, its expression is evaluated, and its value is returned.
- All of the code must appear on one line and a lambda cannot include a selection statement, because selection statements are not expressions.

```
# Lambda function to square a number
square = lambda x: x**2

result = square(4)
print("Result:", result)  #output Result: 16.
```

Higher-Order Functions

```python
# Using lambda expressions with higher-order functions
numbers = [1, 2, 3, 4]

# Using a lambda function to square each element
squared_numbers = list(map(lambda x: x**2, numbers))
print(squared_numbers)  # Output: [1, 4, 9, 16]

# Using lambda with reduce
sum_result = reduce(lambda x, y: x + y, numbers)
print(sum_result)  # Output: 10
```

Higher-Order Functions

Creating Jump Tables

- Jump tables are dictionaries of functions where the keys are command names, and the values are associated functions.
- This approach simplifies command processors, especially in cases where there are many commands.

Higher-Order Functions

```
# Creating a jump table for a command processor
def insert():
    print("Inserting...")

def replace():
    print("Replacing...")

def remove():
    print("Removing...")
# Setup jump table
jump_table = {}
jump_table['1'] = insert
jump_table['2'] = replace
jump_table['3'] = remove

# Function to run commands
def run_command(command):
    jump_table[command]()
# Example usage
run_command('2')  # Output: Replacing...
```

Higher-Order Functions

```python
# Define functions for different cases
def case1():
    print("Executing case 1")
def case2():
    print("Executing case 2")
def default_case():
    print("Executing default case")
# Create a jump table using a dictionary
jump_table = {
    1: case1,
    2: case2,
}
def jump(input_value):
    # Use get() method to retrieve the function for the given key
    # If the key is not found, execute default_case
    jump_table.get(input_value, default_case)()
# Example usage
jump(2)
jump(4)  # This will execute the default_case
```

Exercise

1. Write a Python function to check whether a number is in a given range.
2. Write a Python function that accepts a string and calculate the number of upper case letters and lower case letters. (use isupper() and islower() methods)
3. Write a Python function that takes a number as a parameter and check the number is prime or not

Exercise

4. Write a Python function to check whether a number is perfect or not.
   - In number theory, a perfect number is a positive integer that is equal to the sum of its proper positive divisors, that is, the sum of its positive divisors excluding the number itself. Equivalently, a perfect number is a number that is half the sum of all of its positive divisors (including itself).
   - $1 + 2 + 3 = 6 \textit{ and } (1 + 2 + 3 + 6)/2 = 6$
   - $28 = 1 + 2 + 4 + 7 + 14 \textit{ and } (1 + 2 + 4 + 7 + 14 + 28)/2$
5. Write a Python function that checks whether a passed string is palindrome or not
6. Write a Python function to calculate the factorial of a number using recursion
7. Write a Python function to print the fibonacci series using recursion

Lists

List

- A list is a sequence of data values called items or elements.
- An item can be of any type. Each of the items in a list is ordered by position.
- Each item in a list has a unique index that specifies its position.
- The index of the first item is 0, and the index of the last item is the length of the list minus 1.
- In Python, a list is written as a sequence of data values separated by commas.
- The entire sequence is enclosed in square brackets [and].

Lists

List

- A list is a sequence of data values called items or elements.
- An item can be of any type. Each of the items in a list is ordered by position.
- Each item in a list has a unique index that specifies its position.
- The index of the first item is 0, and the index of the last item is the length of the list minus 1.
- In Python, a list is written as a sequence of data values separated by commas.
- The entire sequence is enclosed in square brackets [and].

```
[1951, 1969, 1984]                      # A list of integers
["apples", "oranges", "cherries"]       # A list of strings
[]                                      # An empty list
```

Lists

List Properties

- Lists are ordered
- Accessed by index
- Lists can contain any sort of object
- Lists are changeable (mutable)

List Literals and Basic Operators

Creating a List Inside Another List

- You can also use other lists as elements in a list, thereby creating a list of lists.

```
list_of_lists = [[5, 9], [541, 78]]
```

  - Here, list_of_lists is a list containing two inner lists:
    - The first inner list: [5, 9]
    - The second inner list: [541, 78]
- To access elements in this list of lists, you would use double indexing.

```
element = list_of_lists[0][1]
# Accesses the element 9 in the first inner list
```

List Literals and Basic Operators

```python
# Function to create a matrix by taking user input
def create_matrix(rows, cols):
    matrix = []
    for i in range(rows):
        row = []
        for j in range(cols):
            # Taking user input for each element
            element = int(input(f"Enter element at position ({i+1}, {j
    +1}): "))
            row.append(element)
        matrix.append(row)
    return matrix

# Function to display a matrix
def display_matrix(matrix):
    for row in matrix:
        for element in row:
            print(element, end=" ")
        print()  # Move to the next line after each row
```

List Literals and Basic Operators

```
# Example usage
rows = int(input("Enter the number of rows: "))
cols = int(input("Enter the number of columns: "))

# Create a matrix
my_matrix = create_matrix(rows, cols)

# Display the matrix
print("Matrix:")
display_matrix(my_matrix)
```

List Literals and Basic Operators

Evaluation of List Literals

- When creating a list, the Python interpreter evaluates each element.
  - If an element is a number or a string, the literal itself is included in the resulting list.
  - However, if the element is a variable or an expression, the value of the variable or the result of the expression is included in the list.

```
import math
x = 2
result_list = [x, math.sqrt(x)]
# Output: [2, 1.4142135623730951]
```

List Literals and Basic Operators

Building Lists with Functions

- Lists can be constructed using functions like range() and list().
- The list() function can build a list from any iterable sequence of elements.

```
first = [1, 2, 3, 4]
second = list(range(1, 5))
# Output: [1, 2, 3, 4]
```

List Literals and Basic Operators

List Operations and Functions

- Length (len())
- Subscript Operator ([])
- Concatenation (+)
- Equality (==)

List Literals and Basic Operators

List Operations and Functions

- Length (len()): Determines the number of elements in a list.

```
# Length
my_list = [10, 20, 30, 40, 50]
length_of_list = len(my_list)
print("Length of the list:", length_of_list)
```

- Subscript Operator ([]): Accesses elements by index, similar to strings.

```
# Subscript Operator
fruits = ["apple", "orange", "banana", "grape"]
second_fruit = fruits[1]
print("Second fruit:", second_fruit)

#Output
Second fruit: orange
```

List Literals and Basic Operators

List Operations and Functions

- Concatenation (+): Combines two lists.

```
# Concatenation
list1 = [1, 2, 3]
list2 = [4, 5, 6]
combined_list = list1 + list2
print("Combined list:", combined_list)

#Output
Combined list: [1, 2, 3, 4, 5, 6]
```

- Equality (==): Compares if two lists are equal.

```
# Equality
first_list = [1, 2, 3]
second_list = [1, 2, 3]
are_lists_equal = first_list == second_list
print("Are lists equal?", are_lists_equal)
#Output
Are lists equal? True
```

List Literals and Basic Operators

Printing Lists:

- The print() function can display the contents of a list. Using a for loop with end=" " removes brackets and commas when printing elements in a list.

```
for number in [1, 2, 3, 4]:
    print(number, end=" ")
# Output: 1 2 3 4
```

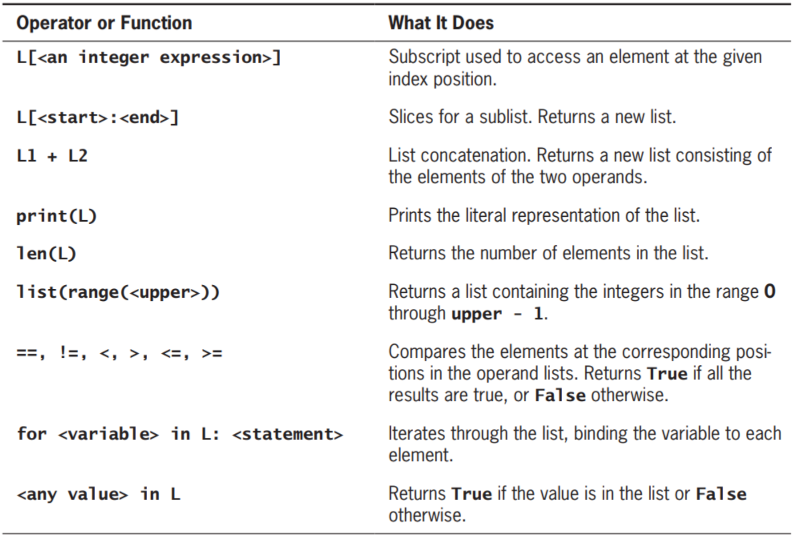
List Literals and Basic Operators

Using the in Operator:

- The in operator checks for the presence or absence of a given element in a list.

```
3 in [1, 2, 3]  # Output: True
0 in [1, 2, 3]  # Output: False
```

List Literals and Basic Operators



Operator or Function	What It Does
`L[<an integer expression>]`	Subscript used to access an element at the given index position.
`L[<start>:<end>]`	Slices for a sublist. Returns a new list.
`L1 + L2`	List concatenation. Returns a new list consisting of the elements of the two operands.
`print(L)`	Prints the literal representation of the list.
`len(L)`	Returns the number of elements in the list.
`list(range(<upper>))`	Returns a list containing the integers in the range **0** through **upper - 1**.
`==, !=, <, >, <=, >=`	Compares the elements at the corresponding positions in the operand lists. Returns **True** if all the results are true, or **False** otherwise.
`for <variable> in L: <statement>`	Iterates through the list, binding the variable to each element.
`<any value> in L`	Returns **True** if the value is in the list or **False** otherwise.

Figure: Some operators and functions used with lists

List Literals and Basic Operators

Replacing an Element in a List

- In Python, lists are mutable, which means their elements can be modified, replaced, or updated.
- This flexibility sets them apart from immutable data types like strings.
- Consider a list example initially containing [1, 2, 3, 4].
- We can use the subscript operator to replace an element at a specific position.
- In this case, we replace the element at index 3 with the value 0.

```
example = [1, 2, 3, 4]
print(example)  # Output: [1, 2, 3, 4]

example[3] = 0
print(example)  # Output: [1, 2, 3, 0]
```

List Literals and Basic Operators

Using a For Loop to Replace Elements:

- In the next example, we replace each number in a list (numbers) with its square.
- This is achieved by iterating over the indices using a for loop.

```python
numbers = [2, 3, 4, 5]
print(numbers)  # Output: [2, 3, 4, 5]

for index in range(len(numbers)):
    numbers[index] = numbers[index] ** 2

print(numbers)  # Output: [4, 9, 16, 25]
```

List Literals and Basic Operators

String Splitting and Uppercasing:

- Another example demonstrates using the split() method on a string to create a list of words. Subsequently, a for loop is used to convert each word to uppercase.

```
sentence = "This example has five words."
words = sentence.split()
print(words)  # Output: ['This','example', 'has','five','words.']

for index in range(len(words)):
    words[index] = words[index].upper()

print(words)  # Output: ['THIS','EXAMPLE','HAS','FIVE','WORDS.']
```

List Literals and Basic Operators

List Methods for Inserting and Removing Elements

- In Python, the list type provides several methods for efficiently inserting and removing elements.
- These methods are essential for dynamically modifying the content of a list

List Methods for Inserting and Removing Elements

insert(index, element)

- The insert method is used to insert a new element at a specified index in the list.
- If the index is less than the length of the list, the new element is placed before the existing element at that index, shifting other elements to the right.
- If the index is greater than or equal to the length, the new element is added to the end.

```
example = [1, 2]
example.insert(1, 10)
# Output: [1, 10, 2]

example.insert(3, 25)
# Output: [1, 10, 2, 25]
```

List Methods for Inserting and Removing Elements

append(element)

- The append method adds a new element to the end of the list.
- It is a convenient way to extend a list with a single element.

```
example = [1, 2]
example.append(3)
# Output: [1, 2, 3]
```

List Methods for Inserting and Removing Elements

extend(iterable)

- The extend method is similar to append, but it takes an iterable (e.g., another list) and adds its elements to the end of the list.

```
example = [1, 2]
example.extend([11, 12, 13])
# Output: [1, 2, 11, 12, 13]
```

List Methods for Inserting and Removing Elements

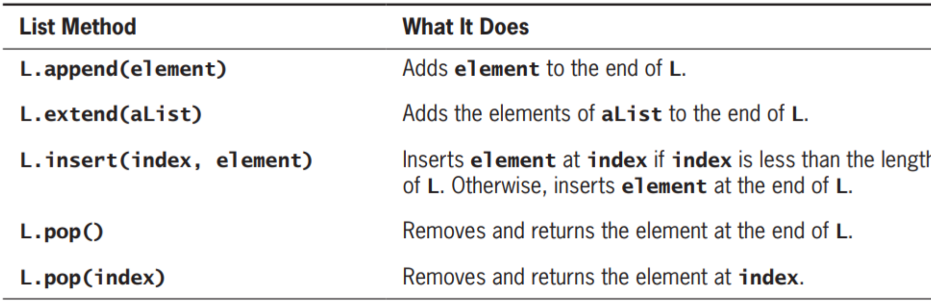
pop([index])

- The pop method removes and returns an element at the specified index.
- If no index is provided, it removes and returns the last element. The list is modified after the operation.

```
example = [1, 2, 10, 11, 12, 13]
last_element = example.pop()
# last_element: 13, example: [1, 2, 10, 11, 12]

first_element = example.pop(0)
# first_element: 1, example: [2, 10, 11, 12]
```

List Methods for Inserting and Removing Elements



List Method	What It Does
`L.append(element)`	Adds **`element`** to the end of **`L`**.
`L.extend(aList)`	Adds the elements of **`aList`** to the end of **`L`**.
`L.insert(index, element)`	Inserts **`element`** at **`index`** if **`index`** is less than the lengt of **`L`**. Otherwise, inserts **`element`** at the end of **`L`**.
`L.pop()`	Removes and returns the element at the end of **`L`**.
`L.pop(index)`	Removes and returns the element at **`index`**.

Figure: List methods for inserting and removing elements

Searching a List

Searching a List

- In Python, searching for an element in a list involves using the in operator to determine the presence of the element.
- However, if you need to find the position (index) of an element within a list, the index method is employed.
- It's important to note that the index method raises an exception when the target element is not found.
- To handle this, programmers often combine the in operator with the index method.

Searching a List

Searching a List

- Consider a list aList = [34, 45, 67] and a target element target = 45. The goal is to find the position of the target element within the list.

```
aList = [34, 45, 67]
target = 45

if target in aList:
    position = aList.index(target)
    print(position)
else:
    print(-1)
```

- In this example:
  - The in operator checks if the target element (45) is present in the list aList.
  - If the target is found, the index method is used to determine the position of the target element.
  - If the target is not found, the else block is executed, and -1 is printed.

Sorting a List

Sorting a List

- In Python, the list type provides a method called sort that allows you to impose a natural ordering on the elements of a list.
- This method mutates the list by rearranging its elements in ascending order.
- Consider a list example = [4, 2, 10, 8]. We want to sort this list in ascending order using the sort method.

```
example = [4, 2, 10, 8]
print(example)  # Output: [4, 2, 10, 8]

example.sort()
print(example)  # Output: [2, 4, 8, 10]

# Sorting the list in descending order using sort()
example.sort(reverse=True)
print(example)  # Output: [10, 8, 4, 2]
```

List Comprehension

List Comprehension

- List comprehensions in Python are concise and expressive ways to create new lists by applying an expression to each element of an existing iterable (e.g., a list, tuple, or range).
- They offer a more compact and readable alternative to traditional for-loops when constructing lists.
- Basic Syntax:

```
new_list = [expression for element in iterable if condition]
```

  - expression: The operation or transformation to be applied to each element.
  - element: The variable representing each element in the iterable.
  - iterable: The existing collection of elements to iterate over.
  - condition (optional): An additional filtering condition to include only certain elements.

List Comprehension

- Example 1: Squares of Numbers Using a For Loop

```
numbers = [1, 2, 3, 4]
squares = [n**2 for n in numbers]
print(squares)
# Output: [1, 4, 9, 16]
```

- Example 2: Finding Common Numbers from Two Lists

```
first = [1, 2, 3, 5]
second = [2, 4, 5, 7]
common = [x for x in first if x in second]
print(common)
# Output: [2, 5]
```

- In the second example, the list comprehension creates a new list (common) by iterating over the elements of the first list and including only those elements that are present in the second list.

Aliasing and Side Effects

Aliasing and Side Effects

- Aliasing occurs when two or more variables refer to the same underlying object or data structure.
- This means that changes made to the object through one variable will be reflected when accessing the object through another variable.

```
# Aliasing with Lists
first = [10, 20, 30]
second = first  # 'second' is now an alias for the same list object
     as 'first'

# Modifying the list through one alias affects the other alias
first[1] = 99
print(second)  # Output: [10, 99, 30]
```

- In this example, first and second are aliases for the same list object.
- When we modify the list through first, the change is visible when accessing the list through second. This is aliasing in action

Aliasing and Side Effects

Preventing Aliasing:

- To avoid aliasing and its potential side effects, you can create a new object and copy the contents of the original object into it.

```
# Preventing Aliasing by Creating a Copy
third = list(first)  # 'third' is a new list with the same elements
     as 'first'

# Modifying 'first' does not affect 'third'
first[1] = 100
print(first)  # Output: [10, 100, 30]
print(third)  # Output: [10, 99, 30]
```

  - In this case, modifying the first list does not affect the third list because they are two separate objects, not aliases.
  - The list() function is used to create a new list with the same elements as the original list, preventing aliasing.

Using a List to Find the Median of a Set of Numbers

Example: Using a List to Find the Median of a Set of Numbers

```
# Requesting the filename from the user
fileName = input("Enter the filename: ")

# Opening the file in read mode
f = open(fileName, 'r')

# Initializing an empty list to store numbers
numbers = []

# Input the text, convert it to numbers, and add the numbers to the list
for line in f:
    words = line.split()
    for word in words:
        numbers.append(float(word))
```

Using a List to Find the Median of a Set of Numbers

```
# Calculate the midpoint of the sorted list
numbers.sort()
midpoint = len(numbers) // 2

# Print the median
print("The median is", end=" ")
if len(numbers) % 2 == 1:
    print(numbers[midpoint])
else:
    print((numbers[midpoint] + numbers[midpoint - 1]) / 2)
f.close()
```

Exercise

Exercise

1. Write a Python program to add 2 matrices
2. Write a Python program to find the transpose of a matrix
3. Write a Python program to multiply 2 matrices

Tuples

Tuples

- Tuples in Python are indeed immutable sequences, and they are defined using parentheses.
- **Creating Tuples:**

```
t=()          #empty tuple
fruits = ("apple", "banana")
meats = ("fish", "poultry")
```

  - Here, fruits and meats are tuples defined using parentheses.
- Concatenating Tuples:

```
food = meats + fruits
```

  - The food tuple is created by concatenating the meats and fruits tuples.

Tuples

Converting List to Tuple:

```
veggies = ["celery", "beans"]
tuple(veggies)
```

- The tuple() function is used to convert the list veggies into a tuple.

Operators and Functions:

- Most of the operators and functions that work with lists can also be used with tuples.
- For example, you can use indexing, slicing, and other common operations.

Tuples

Accessing Elements in a Tuple

- Accessing elements in a tuple is done using index notation.
- In Python, tuples are zero-indexed, meaning the index of the first element is 0, the second element is 1, and so on.
- You can access individual elements or use slicing to access multiple elements. Here's a brief overview:

```
# Creating a tuple
my_tuple = (10, 20, 30, "apple", "orange", "banana")

# Accessing individual elements
first_element = my_tuple[0]
second_element = my_tuple[1]

# Accessing elements using negative indexing (counting from the end)
last_element = my_tuple[-1]  # equivalent to my_tuple[5]
second_last_element = my_tuple[-2]  # equivalent to my_tuple[4]
```

Tuples

```python
# Slicing to access a range of elements
slice_of_tuple = my_tuple[2:5]  # Elements from index 2 to 4 (not
    including 5)
all_elements_after_index_2 = my_tuple[2:]  # Elements from index 2 to
    the end

# Printing the results
print("Tuple:", my_tuple)
print("First element:", first_element)
print("Second element:", second_element)
print("Last element:", last_element)
print("Second last element:", second_last_element)
print("Slice of tuple:", slice_of_tuple)
print("All elements after index 2:", all_elements_after_index_2)
```

Tuples

In this example, my_tuple is a tuple containing both integers and strings. Various methods of accessing elements are demonstrated, including positive and negative indexing as well as slicing.

```
#Tuple: (10, 20, 30, 'apple', 'orange', 'banana')
#First element: 10
#Second element: 20
#Last element: banana
#Second last element: orange
#Slice of tuple: (30, 'apple', 'orange')
#All elements after index 2: (30, 'apple', 'orange', 'banana')
```

Tuples

Slicing

- Slicing a tuple in Python allows you to extract a portion of the tuple by specifying a range of indices.
- The basic syntax for slicing is tuple[start:stop], where start is the index of the first element you want to include, and stop is the index of the first element you want to exclude.

Tuples

```
# Creating a tuple
my_tuple = (10, 20, 30, 40, 50, 60, 70, 80, 90, 100)
# Slicing examples
slice_1 = my_tuple[2:6]   # Elements from index 2 to 5 (not including 6)
slice_2 = my_tuple[:4]     # Elements from the beginning to index 3
slice_3 = my_tuple[7:]     # Elements from index 7 to the end
slice_4 = my_tuple[1:9:2]   # Every second element from index 1 to 8

# Negative indexing can also be used
slice_5 = my_tuple[-5:-2] # Elements from index -5 to -3 (not including
    -2)

# Printing the slices
print("Original Tuple:", my_tuple)
print("Slice 1:", slice_1)
print("Slice 2:", slice_2)
print("Slice 3:", slice_3)
print("Slice 4:", slice_4)
print("Slice 5:", slice_5)
```

Tuples

Special Case: Tuple of One Element:

```
badSingleton = (3)
goodSingleton = (3,)
```

- When creating a tuple with a single element, you need to be careful. In the example, badSingleton is assigned the value 3, not a tuple.
- To create a tuple with a single element, you should include a comma after the element, as shown in goodSingleton.

Tuples

Changing or Deleting a Tuple

- Tuples are immutable
- Elements of a tuple cannot be changed once it has been assigned
- If the element is itself a mutable datatype like list, its nested items can be changed
- The del statement is used to delete the entire tuple.

Tuples

```
# Creating a tuple
original_tuple = (1, 2, [3, 4], 5)

# Attempting to change an element in the original tuple
# Uncommenting the line below would result in an error:
# original_tuple[1] = 99  # TypeError: 'tuple' object does not support
    item assignment

# Changing a nested item in the original tuple
original_tuple[2][0] = 99

# Deleting a tuple
del original_tuple

# Uncommenting the line below would result in an error since the tuple
    is deleted
# print(original_tuple)  # NameError: name 'original_tuple' is not
    defined
```

Tuples

Converting lists into tuples and tuples into lists

```
>>>L=[10,20,30,40]
>>>T=tuple(L)
>>>T (10,20,30,40)
>>> T=(10,20,30)
>>> L=list(T)
>>> L
[10, 20, 30]
```

Tuples

Program to read numbers and find minimum, maximum and sum using Tuple

```
n=int(input("Enter how many numbers......"))
print("Enter numbers")
t=tuple()
for i in range(n):
    x=int(input())
    t=t+(x,)
print("minimum=",min(t))
print("maximum=",max(t))
print("sum=",sum(t))
```

Set

Set

- Sets are used to store multiple items in a single variable
- Set is an unordered and unindexed collection of items
- Every set element is unique (no duplicates)
- Set elements are immutable (cannot be changed)
- Set itself is mutable
- Sets can be used to perform mathematical set operations

Set

Creating Sets:

- Sets can be created using curly braces {} separated by commas or using the built-in set() constructor.

```
my_set = {1, 2, 3}
another_set = set([3, 4, 5])
```

Modifying Sets:

- Sets are mutable, and indexing has no meaning. You cannot access or change an element of a set using indexing or slicing.
- **Adding Elements:**
  - add(): Adds a single element.
  - update(): Adds multiple elements and can take tuples, lists, strings, or other sets.

```
my_set.add(4)
another_set.update([5, 6, 7])
```

Set

Removing Elements

- Elements can be removed using discard() and remove() methods.
  - discard(): Leaves the set unchanged if the element is not present.
  - remove(): Raises an error if the element is not present.

```
my_set.discard(2)
another_set.remove(5)
```

Pop and Clear:

- pop(): Removes and returns an arbitrary element from the set (since sets are unordered, there's no guarantee which element will be popped).
- clear(): Removes all elements from the set.

```
popped_element = my_set.pop()
another_set.clear()
```

Set

Set Operations

- Union
- Intersection
- Difference
- Symmetric difference

Set

```python
# Example sets
set_A = {1, 2, 3, 4, 5}
set_B = {3, 4, 5, 6, 7}

# Union: elements that are in set_A or set_B or both
union_result = set_A.union(set_B)
# Alternatively, you can use the | operator for union
# union_result = set_A | set_B

# Intersection: elements that are common to both set_A and set_B
intersection_result = set_A.intersection(set_B)
# Alternatively, you can use the & operator for intersection
# intersection_result = set_A & set_B

# Difference: elements that are in set_A but not in set_B
difference_result = set_A.difference(set_B)
# Alternatively, you can use the - operator for difference
# difference_result = set_A - set_B
```

Set

```
# Symmetric Difference: elements that are in set_A or set_B but not both
symmetric_difference_result = set_A.symmetric_difference(set_B)
# Alternatively, you can use the ^ operator for symmetric difference
# symmetric_difference_result = set_A ^ set_B

# Printing the results
print("Set A:", set_A)
print("Set B:", set_B)
print("Union:", union_result)
print("Intersection:", intersection_result)
print("Difference:", difference_result)
print("Symmetric Difference:", symmetric_difference_result)
```

Set

- In this example:
    - set_A contains elements {1, 2, 3, 4, 5}.
    - set_B contains elements {3, 4, 5, 6, 7}.
    - The set operations are then performed:
    - Union: Elements that are in set_A or set_B or both. Result: {1, 2, 3, 4, 5, 6, 7}
    - Intersection: Elements that are common to both set_A and set_B. Result: {3, 4, 5}
    - Difference: Elements that are in set_A but not in set_B. Result: {1, 2}
    - Symmetric Difference: Elements that are in set_A or set_B but not both. Result: {1, 2, 6, 7}

Set

Method	Description
add()	Adds an element to the set
clear()	Removes all elements from the set
copy()	Returns a copy of the set
difference()	Returns the difference of two or more sets as a new set
difference_update()	Removes all elements of another set from this set
discard()	Removes an element from the set if it is a member. (Do nothing if the element is not in the set)
intersection()	Returns the intersection of two sets as a new set
intersection_update()	Updates the set with the intersection of itself and another

Set

Method	Description
`isdisjoint()`	Returns `True` if two sets have a null intersection
`issubset()`	Returns `True` if another set contains this set
`issuperset()`	Returns `True` if this set contains another set
`pop()`	Removes and returns an arbitrary set element. Raises `KeyError` if the set is empty
`remove()`	Removes an element from the set. If the element is not a member, raises `KeyError`

Set

Method	Description
`symmetric_difference()`	Returns the symmetric difference of two sets as a new set
`symmetric_difference_update()`	Updates a set with the symmetric difference of itself and another
`union()`	Returns the union of sets in a new set
`update()`	Updates the set with the union of itself and others

Set

Frozen set

- Elements cannot be changed once assigned

```
# Creating a frozenset
frozen_set = frozenset([1, 2, 3, 4, 5])

# Attempting to add an element (will result in an AttributeError)
# Uncommenting the line below would result in an error:
# frozen_set.add(6)

# Attempting to remove an element (will result in an AttributeError
    )
# Uncommenting the line below would result in an error:
# frozen_set.remove(3)

# Printing the frozenset
print("Frozen Set:", frozen_set)
```

Datetime

Datetime

- The datetime module provides functionality for working with dates and times.
- **Displaying the Current Date:**

```
import datetime
x = datetime.datetime.now()
print(x)
```

  - This code imports the datetime module, creates a datetime object representing the current date and time using the now() method, and then prints the result.

Datetime

Creating Date Objects:

```
import datetime
x = datetime.datetime(2020, 5, 17)
print(x)
```

- This code creates a datetime object representing the date May 17, 2020, and then prints the result.
- The datetime class constructor is used with the year, month, and day as parameters.

Optional Parameters and Default Values:

- The datetime() class can take additional parameters for time and timezone (hour, minute, second, microsecond, tzone), but they are optional.
- If not provided, they have default values (hour, minute, second, microsecond are set to 0, and timezone is set to None).

Datetime

Formatting Date Objects:

```
import datetime
x = datetime.datetime(2018, 6, 1)
print(x.strftime("%B"))
```

- This code creates a datetime object representing the date June 1, 2018, and then uses the strftime() method to format the date.
- The %B in the format string represents the full month name. The result is the printed full month name: "June."

Date Formatting

Format Code	Description
%Y	Year with century
%y	Year without century
%m	Month
%B	Full month name
%b	Abbreviated month name

Date Formatting

Format Code	**Description**
%d	Day of the month
%A	Full weekday name
%a	Abbreviated weekday name
%H	Hour (00 to 23)
%I	Hour (01 to 12)

Date Formatting

Format Code	Description
%p	AM or PM
%M	Minute
%S	Second
%f	Microsecond
%z	UTC offset

Date Formatting

Format Code	Description
%Z	Time zone name
%j	Day of the year
%U	Week number (Sunday as first day)
%W	Week number (Monday as first day)
%c	Locale's date and time representation

Date Formatting

```
from datetime import datetime

# Create a datetime object
dt = datetime.utcnow()

# Format the datetime with UTC offset using %z
formatted_time = dt.strftime("%Y-%m-%d %H:%M:%S")

print(formatted_time)

#Output
2024-02-24 15:16:11
```

Dictionary

Dictionary

- Dictionaries organize information by association, not position.
- A dictionary organizes information by pairing keys with corresponding data values, such as names and phone numbers.
- It allows storage of data in key:value pairs, like "Anton": 8182929121, "Jose": 9792129212.
- Dictionaries are unordered, can be modified, and do not allow duplicate keys.
- Keys can be any immutable data type, such as strings, numbers, or tuples.

Dictionary

Dictionary Literals:

- Written as a sequence of key/value pairs enclosed in curly braces.
- Example dictionaries include a phone book and personal information.

```
phone_book = {"Savannah": "476-3321", "Nathaniel": "351-7743"}
personal_info = {"Name": "Molly", "Age": 18}
```

Adding Keys and Replacing Values:

- Use the subscript operator ([]) to add new key/value pairs or replace values.

```
info = {}
info["name"] = "Sandy"
info["occupation"] = "hacker"
```

Dictionary

Accessing Values:

- **Use the subscript** to obtain the value associated with a key.
- Use in operator to check if a key exists.
- **Use the get method** to access a value with a default value if the key is not present.

```
info = {}
info["name"] = "Sandy"
info["occupation"] = "hacker"
print(info["name"])  # Outputs: 'Sandy'

if "job" in info:
    print(info["job"])

print(info.get("job", None))  # Outputs: None
```

Dictionary

Removing Keys:

- Use the pop method to delete an entry from a dictionary by providing the key.
- The method can take an optional default value to return if the key is not present.

```
print(info.pop("job", None))  # Outputs: None
print(info.pop("occupation"))  # Outputs: 'manager'
```

Dictionary

Traversing a Dictionary:

- Use a for loop to iterate over the keys of a dictionary.
- Alternatively, use the items() method to access key-value pairs.

```
# Sample dictionary
grades = {'Alice': 90, 'Bob': 85, 'Charlie': 92}
for key in grades:
    print(key, grades[key])

for key, value in grades.items():
    print(key, value)

#Output
Alice 90
Bob 85
Charlie 92
```

Dictionary

To read a dictionary with key-value pairs from the keyboard:

```
# Initialize an empty dictionary
my_dict = {}

# Get the number of key-value pairs
num_pairs = int(input("Enter the number of key-value pairs: "))

# Input key-value pairs from the user
for i in range(num_pairs):
    key = input("Enter key: ")
    value = input("Enter value: ")

    # Add the key-value pair to the dictionary
    my_dict[key] = value

# Display the resulting dictionary
print("Dictionary:", my_dict)
```

Dictionary Operations - Frame 1

Operation	**Description**
`len(d)`	Returns the number of entries in `d`.
`d[key]=value`	Used for inserting a new key, replacing a value, or obtaining a value at an existing key.

Dictionary Operations - Frame 2

Operation	Description
`d.get(key [, default])`	Returns the value if the key exists or returns the default if the key does not exist. Raises an error if the default is omitted and the key does not exist.
`d.pop(key [, default])`	Removes the key and returns the value if the key exists or returns the default if the key does not exist. Raises an error if the default is omitted and the key does not exist.

Dictionary Operations - Frame 3

Operation	**Description**
`list(d.keys())`	Returns a list of the keys.
`list(d.values())`	Returns a list of the values.
`list(d.items())`	Returns a list of tuples containing the keys and values for each entry.
`d.clear()`	Removes all the keys.

Exercise

1. Write a Python script to sort (ascending and descending) a dictionary by value
2. Write a Python script to concatenate 2 dictionaries to create a new one
3. Write a Python script to check whether a given key already exists in a dictionary
4. Write a Python script to generate and print a dictionary that contains a number (between 1 and n) in the form (x, x*x)
5. Write a Python program to multiply all the items in a dictionary
6. Write a Python program to remove a key from a dictionary
7. Write a Python program to sort a dictionary by key
8. Write a Python program to get the sum of all values, the 3 highest values and the 3 lowest values in a dictionary
9. Write a Python program to convert a list into a nested dictionary of keys

Exercise

10 Write a Python program to create a dictionary of keys x, y, and z where each key has as value a list from 11- 20, 21-30, and 31-40 respectively. Access the fifth value of each key from the dictionary.

Output

```
{'x': [11, 12, 13, 14, 15, 16, 17, 18, 19],
'y': [21, 22, 23, 24, 25, 26, 27, 28, 29],
'z': [31, 32, 33, 34, 35, 36, 37, 38, 39]}
15
25
35
```