LABORATORY MANUAL

of

UCEST105: ALGORITHMIC THINKING WITH PYTHON (S1)

DEPARTMENT OF ELECTRONICS AND COMMUNICATION ENGINEERING

COLLEGE OF ENGINEERING

TRIVANDRUM

DEPARTMENT OF ELECTRONICS AND COMMUNICATION ENGINEERING

COLLEGE OF ENGINEERING

TRIVANDRUM

CERTIFICATE

This is a controlled document of the department of Electronics and Communication Engineering of the College of Engineering, Trivandrum. No part of this can be reproduced in any form by any means without the prior written permission of the Head of the Department, Electronics and Communication Engineering, College of Engineering, Trivandrum. This is prepared as per 2024 KTU B.Tech scheme.

UCEST105: Algorithmic Thinking with Python

Course Code	UCEST105	CIE Marks	40
Teaching Hours/Week (L: T:P: R)	3:0:2:0	ESE Marks	60
Credits	4	Exam Hours	2 Hrs. 30 Min.
Prerequisites (if any)	None	Course Type	Theory

Course Objectives:

- To provide students with a thorough understanding of algorithmic thinking and its practical applications in solving real-world problems..
- To explore various algorithmic paradigms, including brute force, divide-and-conquer, dynamic programming, and heuristics, in addressing and solving complex problems.

Course Outcomes		
CO#	**Description**	K-Level
CO1	Develop algorithmic thinking to analyze problems and implement step-by-step computational solutions using Python.	K3
CO2	Represent algorithms using pseudocode and flowcharts, and analyze their efficiency to select appropriate solutions for a given problem.	K4
CO3	Design modular programs using functions and data abstraction principles for real-world applications.	K3
CO4	Develop programs to implement common algorithms such as searching, sorting, recursion, and data manipulation using Python.	K3
CO5	Collaborate effectively in teams to plan, implement, test, and present a course project that demonstrates algorithmic thinking and problem-solving using Python.	K4

Contents (with COs)

Experiment 1: Getting Started, Input/Output, and Expressions

Aim

Install Python 3, confirm the environment setup, and master fundamental concepts of variables, user input, standard output, and basic arithmetic expressions, including type conversion.

Theory

A computer program is a finite sequence of instructions designed to perform a specified task. In Python, execution follows a **sequential flow model**, where statements are processed from top to bottom.

Variables and Dynamic Typing: Python variables are symbolic names that refer to objects in memory. Unlike static languages, Python uses **dynamic typing**, meaning the type of a variable is inferred at runtime and can change during execution.

Input and Type Casting: The built-in function `input()` is the standard mechanism for receiving user data. Crucially, `input()` always returns the data as a **string** (`str`) type. To perform numeric calculations, **explicit type casting** is required using functions like `int()` (for integers) or `float()` (for floating-point numbers). Failure to cast will result in string concatenation or `TypeError`.

Output and Formatting: The `print()` function sends data to the standard output stream. **Formatted String Literals** (f-strings) provide a concise, readable syntax for embedding expressions inside string constants. F-strings support **format specifiers** (e.g., `:.2f`) to control precision, alignment, and representation of numeric outputs.

Arithmetic Expressions: Python supports standard arithmetic operators (`+`, `-`, `*`, `/`). The division operator (`/`) always yields a float, even if the result is an integer.

Design & Implementation

1. Verify Python 3.11 | is installed via `python -version` in the terminal.
2. Use the interactive shell (`python` or `python3`) for testing quick expressions (e.g., `9/5 * 32 + 32`).
3. Write and execute the initial scripts: `hello.py`, `simple_calc.py`, `circle_area.py`, and `temp_convert.py`.
4. **New Program:** Implement `bmi_calc.py` to compute the Body Mass Index (BMI).

Code (Python)

```python
# hello.py
print("Hello, UCEST105!")

# simple_calc.py (Demonstrates all four basic arithmetic
    operations)
a = float(input("Enter first number: "))
```

```python
b = float(input("Enter second number: "))
print(f"sum={a+b} diff={a-b} prod={a*b} quot={a/b:.4f}")

# circle_area.py (Using a constant for PI)
r = float(input("Radius: "))
PI = 3.141592653589793 # Conventionally, constants are in CAPITALS
area = PI * r * r
print(f"Area of circle with r={r} is {area:.4f}")

# temp_convert.py (Fahrenheit = 9/5 * C + 32)
c = float(input("Celsius: "))
f = 9/5 * c + 32
print(f"{c:.1f} C is {f:.1f} F")

# bmi_calc.py (BMI = weight (kg) / (height (m) ** 2))
weight_kg = float(input("Enter weight (kg): "))
height_m = float(input("Enter height (m): "))
bmi = weight_kg / (height_m ** 2)
print(f"Weight: {weight_kg:.1f}kg, Height: {height_m:.2f}m")
print(f"BMI is: {bmi:.2f}")
```

Results

Execution must confirm both raw computation and presentation control. The `simple_calc` and `bmi_calc` programs are critical for verifying correct type conversion.

Table 2: Summary of Experiment 1 Results

Program	Input Example	Observed Output	Concept Verified
`simple_calc.py`	a=10, b=4	sum=14.0 diff=6.0 prod=40.0 quot=2.5000	`float()` conversion, division
`circle_area.py`	Radius: 5.0	Area of circle with r=5.0 is 78.5398	Use of constants, `:.4f` formatting
`temp_convert.py`	Celsius: 20	20.0 C is 68.0 F	Mathematical expression order
`bmi_calc.py`	Weight: 70, Height: 1.75	BMI is: 22.86	Exponent operator (`**`)

Analysis & Inference

All user input is initially a string; using `float()` ensures the integrity of decimal calculations. The `f-string` mechanism provides precise control over the output, as demonstrated by rounding the BMI to two decimal places (`:.2f`). Failure to convert input would result in concatenation (e.g., `"10" + "4"` becoming `"104"`) or a `TypeError`.

Conclusion

The Python environment is successfully configured. Core concepts of basic I/O, variable assignment, and explicit type casting for arithmetic operations have been understood and verified.

—

Experiment 2: Variables, Data Types, and Operators

Aim

Differentiate between Python's core data types (`int`, `float`, `str`, `bool`), and practice all major operator groups: arithmetic, comparison, logical, and augmented assignment.

Theory

Python supports several fundamental data types. The `int` type provides **arbitrary-precision integers**, meaning they can store numbers limited only by available memory, unlike fixed-size integers in many other languages. The `float` type corresponds to IEEE 754 double-precision floating-point numbers.

String Type (`str`): Strings are immutable sequences of Unicode characters. They support **indexing** (accessing a single character by position, starting from 0) and **negative indexing** (starting from -1 for the last character).

Arithmetic Operators: In addition to basic operations, Python includes:

- **Floor Division (`//`):** Divides and rounds the result down to the nearest integer toward negative infinity.
- **Modulo (`%`):** Returns the remainder of the division. The sign of the result is the same as the divisor.
- **Exponentiation (`**`):** Raises the first operand to the power of the second.

Logical Operators: The `and`, `or`, and `not` operators are used to combine boolean values or expressions, crucial for constructing complex control flow conditions. Python also supports **chained comparisons** (e.g., `a < b < c`), which is equivalent to `(a < b) and (b < c)`.

Augmented Assignment Operators: Shorthand operators like `+=`, `-=`, `*=` update a variable by performing an operation on its current value. For example, `x += 5` is equivalent to `x = x + 5`.

Design & Implementation

Write a single script to perform the following demonstrations:

1. Read two numbers and contrast standard division (`/`) with floor division (`//`) and modulo (`%`).
2. Read a word, display its length, first character (`s[0]`), last character (`s[-1]`), repetition (`s * 3`), and concatenation.
3. Test boolean expressions, including the use of `not` and chained comparisons (e.g., `1 < x < 10`).
4. Demonstrate an augmented assignment operator (`+=`) for an accumulator.

Code (Python)

```python
x = float(input("x: "))
y = float(input("y: "))
print("\n--- Arithmetic Operators ---")
print(f"x/y (Standard Division): {x/y:.2f}")
print(f"x//y (Floor Division): {x//y}")
print(f"x%y (Modulo/Remainder): {x%y}")
print(f"x**y (Exponent): {x**y}")

s = input("\nEnter a word: ")
t = input("Enter another word: ")
print("\n--- String Operations ---")
print(f"Length of '{s}': {len(s)}")
print(f"First char (s[0]): {s[0]}, Last char (s[-1]): {s[-1]}")
print(f"Repetition (s * 3): {s * 3}")
print(f"Concatenation: {s + t}")

n = int(input("\nEnter an integer for logic checks: "))
is_positive = n > 0
is_multiple_of_5 = n % 5 == 0
print("\n--- Boolean and Comparison Operators ---")
print(f"Is {n} positive AND a multiple of 5? {is_positive and
    is_multiple_of_5}")
print(f"Is {n} NOT a multiple of 3? {not (n % 3 == 0)}")
print(f"Is 0 < n < 100? {0 < n < 100}") # Chained comparison

# Augmented assignment demonstration
total_sum = 10
print(f"\nInitial total: {total_sum}")
total_sum += 5
print(f"Total after += 5: {total_sum}")
total_sum *= 2
print(f"Total after *= 2: {total_sum}")
```

Results

The execution should clearly distinguish the behavior of division operators and confirm the flexibility of string indexing and the power of logical operators.

Table 3: Key Results from Operator Demonstration

Operation	Input Example	Result	Inference
x=10, y=3	x//y	3.0	Floor division discards the fractional part.
String Indexing	word="Python"	P, n	Indexing starts at 0; negative index is relative to the end.
Logical Test	n=15	True	and requires both conditions to be true.
Chained Comparison	n=15	True	0 < 15 < 100 is syntactically valid and clean.
Augmented Assign	Initial 10	30.0	+= is a shorthand for total_sum = total_sum + 5.

Analysis & Inference

The behavior of `//` (floor division) toward negative infinity should be noted, though often irrelevant for positive numbers. Strings are sequences that support powerful indexing; `s[-1]` is a common idiomatic way to get the last element. The `and` and `or` operators allow programmers to create conditions that accurately model real-world requirements. The successful demonstration of augmented assignment confirms its utility in loop-based tasks.

Conclusion

All major Python data types and operator groups are effectively utilized. A foundational understanding of numerical precision, string sequence properties, and logical flow control is established.

Experiment 3: Selection (if/elif/else) and Simple Programs

Aim

Practice constructing control flow logic using `if`, `elif`, and `else` statements to implement decision-making algorithms based on input conditions.

Theory

Control Flow: Programs rarely execute in a straight line. **Selection** (or branching) is a fundamental control structure that determines which block of code should be executed based on the truth value of a condition.

The `if/elif/else` Construct:

- `if`**:** Evaluates the primary condition. If true, the associated code block is executed, and the interpreter skips the rest of the `elif/else` chain.
- `elif` **(Else If):** Provides an alternative condition. It is checked only if all preceding `if` and `elif` conditions were false. This structure guarantees that only **one** block in the entire chain will execute.
- `else`**:** The default block. It executes only if all preceding `if` and `elif` conditions are false.

Indentation: Python uses **indentation** (typically four spaces) to define code blocks, including those associated with `if/elif/else`. This is mandatory and ensures structural clarity.

Algorithmic Importance of Order: When checking overlapping conditions (e.g., grade ranges), the conditions must be ordered carefully (e.g., from highest score range to lowest) to ensure the logic is correctly applied and the most specific condition is checked first.

Design & Implementation

Write one script containing the three specified tasks and a new task:

1. **Grade Calculator:** Ensure `elif` conditions are ordered from highest to lowest score to correctly assign grades based on overlapping ranges.
2. **Leap Year Test:** Implement the full Gregorian calendar rule using a single, clear compound boolean expression: a year is a leap year if it is divisible by 400, OR if it is divisible by 4 AND not divisible by 100.
3. **Largest of Three Numbers:** Find the maximum value using a sequence of `if` statements, which is a simple form of comparative sorting.
4. **New Program:** `check_triangle.py` - Determine if three sides form a triangle and classify it (Equilateral, Isosceles, Scalene). This requires checking the **Triangle Inequality Theorem** ($a + b > c$, $a + c > b$, and $b + c > a$) first.

Code (Python)

```python
# grade (Order matters: check highest score first)
m = float(input("Marks (0-100): "))
if   m >= 90: g = "A"
elif m >= 80: g = "B"
elif m >= 70: g = "C"
elif m >= 60: g = "D"
else: g = "F"
print("Grade:", g)

# leap year (Compound condition)
y = int(input("Year: "))
is_leap = (y % 400 == 0) or ((y % 4 == 0) and (y % 100 != 0))
print(f"Year {y} is a Leap year: {is_leap}")

# largest of three (Sequential comparison)
a = float(input("a: ")); b = float(input("b: ")); c = float(input("c: "))
largest = a
if b > largest: largest = b
if c > largest: largest = c # Note: No 'elif' needed here as we want to check C regardless
print("Largest:", largest)

# triangle classification
print("\n--- Triangle Classifier ---")
s1 = float(input("Side 1: "))
s2 = float(input("Side 2: "))
s3 = float(input("Side 3: "))

# 1. Check Triangle Inequality Theorem
if s1 + s2 > s3 and s1 + s3 > s2 and s2 + s3 > s1:
        print("These sides CAN form a triangle.")
# 2. Classification
if s1 == s2 and s2 == s3:
        print("Classification: Equilateral (All sides equal)")
elif s1 == s2 or s2 == s3 or s1 == s3:
        print("Classification: Isosceles (Two sides equal)")
else:
        print("Classification: Scalene (No sides equal)")
else:
        print("These sides CANNOT form a triangle (Fails Triangle Inequality).")
```

Results

Testing with boundary and negative cases verifies the robustness of the conditional logic.

Table 4: Test Cases for Selection Logic

Algorithm	Input	Expected/Observed Outcome	Logic Verified
Grade Calc.	Marks: 89.5	B	`elif` execution order
Leap Year	1900	False	`(y % 100 != 0)` part of the rule
Largest	10, 20, 15	Largest: 20.0	Sequential `if` updates the `largest` variable
Triangle	3, 4, 10	CANNOT form a triangle	Triangle Inequality (e.g., $3 + 4 \not> 10$)
Triangle	5, 5, 5	Equilateral	Nested `if/elif/else` for classification

Analysis & Inference

The grade calculator confirms that `elif` guarantees exclusivity, preventing a score of 95 from being incorrectly classified as B. The leap year logic demonstrates the power of combining `and` and `or` operators for compact rule implementation. The triangle classifier showcases the necessity of checking a **prerequisite condition** (inequality theorem) before proceeding to detailed classification, a key algorithmic pattern.

Conclusion

Decision-making logic is correctly implemented using `if/elif/else`. The order and complexity of conditions were managed successfully to solve practical problems.

Experiment 4: Loops (for, while), Ranges, and Patterns

Aim

Utilize `for` and `while` loops to automate repetitive tasks, understand the `range()` function, and practice using loops for numeric accumulation and simple pattern generation.

Theory

Iteration is the ability to execute a block of code repeatedly. Python provides two primary looping constructs: `for` and `while`.

The `for` Loop (Definite Iteration): This loop iterates over the items of any sequence (such as a list, tuple, dictionary, set, or string) or other iterable object.

- **`range()`:** The `range(start, stop, step)` function is commonly used with the `for` loop. It generates a sequence of integers up to, but not including, the `stop` value. It is efficient because it generates numbers on demand, rather than creating a full list in memory.

The `while` Loop (Indefinite Iteration): This loop executes a block of code repeatedly as long as its condition remains true. It is best used when the number of iterations is not known beforehand, such as reading data until a specific **sentinel value** is encountered.

Loop Control and Accumulation:

- **Counters:** Variables used within loops to track the number of iterations performed.
- **Accumulators:** Variables used within loops to collect or aggregate results (e.g., calculating a running sum or product).
- **`break`:** Immediately exits the current loop, regardless of the loop condition.
- **`continue`:** Skips the rest of the current iteration and moves to the next one.

Pattern Generation: Simple patterns are often generated by using a loop (outer loop) to control the number of rows and either an inner loop or string multiplication to control the content of each row.

Design & Implementation

Implement the following tasks in a single script:

1. **Sum of N Integers:** Use a `for` loop and an accumulator variable.
2. **Factorial:** Calculate N! using a `while` loop.
3. **Fibonacci Series:** Print the first N terms, demonstrating efficient state update (`a, b = b, a + b`) which is a common multiple assignment idiom.
4. **Left-aligned Triangle:** Print a star pattern using `for` and string repetition.
5. **New Task:** Implement a **Sentinel-Controlled Loop** using `while True` and a `break` statement to process data until a specific termination value (the sentinel) is entered.

Code (Python)

```python
N = int(input("N (for Sum/Factorial/Fibonacci): "))

# 1. Sum 1..N (Definite iteration with for)
s = 0
for i in range(1, N+1):
        s += i
print(f"Sum of 1 to {N}: {s}")

# 2. Factorial (Indefinite iteration with while)
f = 1
k = 1
while k <= N:
        f *= k  # Accumulator for product
        k += 1  # Counter update
print(f"Factorial of {N}: {f}")

# 3. Fibonacci series (Efficient state management)
a, b = 0, 1
print("Fibonacci series:", end=" ")
for _ in range(N): # Use _ for a throwaway variable if the loop index isn't needed
        print(a, end=" ")
        a, b = b, a + b # Simultaneous update of sequence terms
        print()

# 4. Pattern
rows = int(input("rows for pattern: "))
        for r in range(1, rows+1):
        print("*" * r)

# 5. Sentinel-Controlled Loop
total = 0
print("\n--- Sentinel Loop (Enter 0 to stop) ---")
while True:
        num_str = input("Enter number: ")
try:
        num = int(num_str)
        if num == 0:
                break # Sentinel condition met, exit loop
        total += num
except ValueError:
        print("Invalid input. Please enter an integer.")

print(f"Total sum from sentinel loop: {total}")
```

Results

Displays sum, factorial, Fibonacci numbers, a star pattern, and the total from the sentinel loop.

Table 5: Loop Functionality Test Cases

Task	Input	Output Example	Loop/Control Feature
Sum	N=5	Sum: 15	`range()` and `+=` accumulator
Factorial	N=5	Factorial: 120	`while` loop condition and `*=` accumulator
Fibonacci	N=6	0 1 1 2 3 5	Simultaneous variable assignment
Pattern	rows=3	***	`print()` loop with string multiplication
Sentinel Loop	10, 20, 0	Total sum: 30	`while True` and `break` on sentinel value

Analysis & Inference

The `for` loop is the most concise solution for the sum and pattern tasks because the number of iterations is predefined. The `while` loop for the sentinel task proves necessary when the exit condition depends on data received inside the loop. The Fibonacci update `a, b = b, a + b` is a Python idiom for updating two variables atomically. The absence of integer overflow errors, even for large factorials, highlights Python's arbitrary-precision integers.

Conclusion

Both definite (`for`) and indefinite (`while`) iteration techniques are implemented successfully. Loops are confirmed as the primary mechanism for computational repetition and sequence generation.

Experiment 5: Functions, Parameters, Return Values, and Testing

Aim

Structure code into reusable, modular functions; master parameter passing (positional, keyword, and default), `return` values, and use the `if __name__ == "__main__"` block for automated testing.

Theory

Modularity and Functions: A function is a named, reusable block of code defined using the `def` keyword. It is a core principle of modular programming, enhancing code readability, reducing redundancy, and promoting reuse.

Parameters and Arguments: Parameters are the variables listed inside the function's definition; they receive the values (arguments) passed during a function call. Python supports:

- **Positional Arguments:** Matched to parameters based on their order.
- **Keyword Arguments:** Explicitly named in the call, allowing them to be passed in any order.
- **Default Parameters:** Parameters assigned a standard value in the function signature. If an argument for this parameter is omitted in the call, the default value is used.

Return Values: The `return` statement terminates a function's execution and passes a value (or `None`, if omitted) back to the caller. Functions can return multiple values as a tuple.

Docstrings and Testing:

- **Docstrings:** Multi-line string literals used immediately after the `def` line to document a function's purpose, parameters, and return value.
- `if __name__ == "__main__":` This is the standard entry-point guard. It ensures that code within this block (typically setup, execution examples, or **unit tests**) only runs when the file is executed directly, and not when it is imported as a module into another script.
- `assert`: Used for simple, quick tests. If the expression following `assert` is false, the program halts with an `AssertionError`.

Design & Implementation

1. Create `utils.py` as a module containing all function definitions.
2. Include functions for temperature conversion, absolute value, and palindrome check.
3. **New Function:** Implement `power(base, exp=2)` to demonstrate a **default parameter**.

4. Write simple self-tests using `assert` within `__name__ == "__main__"` in `utils.py`.

5. Create `main.py` to import and use the utility functions, demonstrating both standard and default parameter calls.

Code (Python)

```python
# utils.py
def c_to_f(c):
"""Convert Celsius to Fahrenheit."""
        return 9/5 * c + 32

def abs_val(x):
"""Return absolute value without using built-in abs."""
        return x if x >= 0 else -x # Concise conditional expression

def is_palindrome(s):
"""Return True if string reads the same forwards/backwards (ignores case/spaces)."""
        t = s.replace(" ", "").lower()
        return t == t[::-1] # Check if string equals its reverse slice

def power(base, exp=2):
"""Calculate base raised to the power of exp. Default exponent is 2 (square)."""
        return base ** exp

if __name__ == "__main__":
# Unit tests: Assertions test expected behavior
        print("Running utils self-tests...")
        assert c_to_f(100) == 212.0
        assert abs_val(-10) == 10
        assert is_palindrome("Racecar") is True
        assert power(4) == 16       # Test default parameter (square)
        assert power(2, exp=3) == 8 # Test keyword parameter
        print("All utils self-tests passed.")
```

```python
# main.py
import utils # Import the utility module

print("--- Function Demonstration ---")
c = float(input("Celsius to convert: "))
print(f"Fahrenheit: {utils.c_to_f(c):.1f}")

num = float(input("Number for absolute value: "))
print(f"Absolute value: {utils.abs_val(num)}")

```

```python
phrase = input("Phrase for palindrome check: ")
print(f"'{phrase}' is a palindrome: {utils.is_palindrome(phrase)}"
    )

# Demonstrate default and keyword arguments
b = float(input("Base number: "))
print(f"Square of {b}: {utils.power(b)}") # Uses default exp=2
print(f"Cube of {b}: {utils.power(b, 3)}") # Positional argument 3
```

Results

Running `utils.py` directly should print the success message, confirming function correctness. Running `main.py` verifies successful module import and function usage, including flexible argument passing.

Table 6: Function Usage and Parameter Verification

Function Call	Arguments	Return Value	Concept Verified
`utils.c_to_f(20)`	Positional 20	68.0	Standard `return` value
`utils.power(5)`	5	25	Default parameter `exp=2` used
`utils.power(2, 4)`	Positional 2, 4	16	Default parameter overridden
`is_palindrome("madam")`	"madam"	True	Function logic and string reversal slicing

Analysis & Inference

Separating code into `utils.py` and `main.py` promotes **modularity** and **testability**. The `if __name__ == "__main__"` guard effectively separates the module's test suite from its external usage. The `power` function demonstrates how default parameters make a function more flexible and easier to use for common cases (squaring). The use of `assert` is a simple but effective form of **unit testing**.

Conclusion

Functions are successfully implemented and demonstrated with various parameter types. The program structure uses modules and the main guard for clean organization and basic self-testing.

Experiment 6: Strings Deep Dive (Indexing, Slicing, Methods, Formatting)

Aim

Perform advanced manipulation and analysis of text data using string methods, slicing, searching, replacing, splitting, joining, and advanced formatting controls for professional text output.

Theory

Strings as Sequences: Strings in Python are ordered, immutable sequences. Their sequential nature allows for access to sub-parts of the string through **slicing**.

Slicing Syntax: The syntax is `s[start:stop:step]`.

- Omitting `start` defaults to 0; omitting `stop` defaults to the end.
- A `step` of `-1` (`s[::-1]`) is the idiomatic way to reverse a string.

String Methods for Manipulation: Python provides numerous built-in methods (which do not change the original string due to immutability, but return a new one):

- **Cleaning:** `strip()`, `lower()`, `upper()`.
- **Searching/Replacing:** `find()`, `replace()`.
- **Splitting/Joining:** `split()` breaks a string into a list of substrings (words, typically by whitespace); `join()` concatenates a list of strings into a single string using a specified delimiter.

Advanced Formatting (f-strings): F-strings support the **Format Specification Mini-Language**, providing granular control over output fields:

- **Alignment:** `<` (left-align), `>` (right-align), `^` (center-align), often combined with a width specifier (e.g., `:^10`).
- **Precision:** `.2f` specifies a floating-point number rounded to two decimal places.
- **Separators:** `:,` adds thousands separators (commas).

Design & Implementation

Write a script that processes user-input text for several tasks:

1. Count vowels and words using `lower()` and `split()`.
2. Remove punctuation using `string.punctuation` and a **generator expression** with `join()`.
3. Demonstrate string reversal using the slicing technique `s[::-1]`.
4. Implement title-casing.
5. Format a table row with explicit field widths, alignment, and decimal control.

Code (Python)

```python
import string

text = input("Enter a sentence, including punctuation: ")

# 1. Count Vowels and Words
vowels = set("aeiou") # Use a set for O(1) membership check
vcount = sum(1 for ch in text.lower() if ch in vowels)
wcount = len(text.split())
print(f"\nAnalysis: Vowels={vcount}, Words={wcount}")

# 2. Remove Punctuation and 3. Reverse String
# Generator expression inside join for efficiency
no_punct = "".join(ch for ch in text if ch not in string.
    punctuation)
print(f"No punctuation: '{no_punct}'")
print(f"String reversed (s[::-1]): '{text[::-1]}'")

# 4. Title Case
print(f"Built-in Title Case: {text.title()}")

# 5. Format a Table Row
name = "Dr. Smith"; age = 45; salary = 89123.456
# Format: Name (Left, 15), Age (Center, 5), Salary (Right, 10,
    comma separation, 2 decimals)
print("\n--- Formatted Table Row ---")
print("| Name            | Age |  Salary   |")
print("|-----------------|-----|-----------|")
print(f"| {name:<15} | {age:^5d} | {salary:>10,.2f} |")
```

Results

Displays counts, cleaned text, reversed text, title-cased text, and a professionally formatted row.

Table 7: String Manipulation and Formatting Verification

Input Example	Operation	Observed Output	Technique
`"Hello, World!"`	No Punctuation	Hello World	`string.punctuation` and `join()`
`"stressed"`	Reverse	desserts	`s[::-1]` slicing
`"alice"`	Title Case	Alice	`title()` method
`Salary: 89123.456`	Formatting	89,123.46	`:>10,.2f` specifier (alignment, comma, precision)

Analysis & Inference

The use of a **generator expression** inside `join()` for removing punctuation is efficient as it avoids creating a full intermediate list. The slicing shortcut `s[::-1]` is the Pythonic

way to reverse a sequence. F-string formatting, demonstrated by the salary output (`:,` for thousands separator, `.2f` for precision), is essential for high-quality, readable report generation.

Conclusion

Advanced string operations, including slicing and various methods for text preparation, are mastered. Precise output formatting using f-strings is confirmed.

Experiment 7: Lists and Tuples (Slicing, Methods, Comprehensions)

Aim

Manipulate ordered data collections using Lists (mutable) and Tuples (immutable). Practice common collection methods, slicing, and efficient list construction with comprehensions.

Theory

Ordered Collections: Lists and tuples are both ordered sequences, maintaining the order of elements as they are inserted.

Lists (Mutable): Lists are dynamic arrays, defined by square brackets (`[]`). Their key characteristic is **mutability**, meaning their content can be changed after creation. Common list methods modify the list **in place** (e.g., `append, insert, pop, remove, sort`). Slicing a list returns a **new** list object.

Tuples (Immutable): Tuples are fixed-size sequences, defined by parentheses (`()`). Their primary characteristic is **immutability**, meaning their elements cannot be added, removed, or changed after creation. This makes them suitable for use as data records, function return values, and dictionary keys (provided their contents are also immutable).

List Comprehensions (LC): LCs provide a concise, readable, and often highly efficient way to create a list based on an existing iterable. The general syntax is: `[expression for item in iterable if condition]` They perform the tasks of mapping (transforming elements) and filtering (selecting elements) simultaneously.

Deduplication: Removing duplicates from a list while maintaining order is a common algorithmic challenge often solved by iterating over the list and using a **Set** (an unordered collection of unique elements) to efficiently track which items have already been seen in $O(1)$ time.

Design & Implementation

Tasks within a single script:

1. Create a list, and demonstrate mutability: `append, insert, remove, pop`. Slice the list.

2. Compute statistics (`min, max, avg`) using built-in functions.

3. **Deduplication:** Remove duplicate elements while retaining the original order, utilizing a `set` for $O(1)$ membership checking.

4. Use list comprehensions to create a list of squares (mapping) and a filtered list of even numbers.

5. Demonstrate tuple creation, element access, and the immutability constraint (by attempting a modification).

Code (Python)

```python
nums = [5, 2, 9, 2, 7, 5, 1, 6]
print(f"1. Original list: {nums}")

# 1. List Mutability Demonstration
nums.append(10)       # Adds to end
nums.insert(2, 99)    # Inserts at index 2
nums.remove(2)        # Removes first instance of value 2
removed_val = nums.pop(0) # Removes and returns item at index 0 (5)
print(f"Modified list: {nums}")
print(f"Slice [2:5] (elements at index 2, 3, 4): {nums[2:5]}")

# 2. Statistics
mn = min(nums)
mx = max(nums)
avg = sum(nums) / len(nums)
print(f"\n2. Min: {mn}, Max: {mx}, Avg: {round(avg, 2)}")

# 3. Remove Duplicates preserving order
seen = set() # O(1) lookup
unique = []
for x in nums:
        if x not in seen:
                seen.add(x)
                unique.append(x)
print(f"\n3. Unique elements (order preserved): {unique}")

# 4. List Comprehensions (Map and Filter)
squares = [x*x for x in unique] # Mapping
evens = [x for x in unique if x % 2 == 0] # Filtering
print(f"4. Squares: {squares}")
print(f"   Evens: {evens}")

# 5. Tuples (Immutable Sequence)
person = ("Alice", 25, "New York")
print(f"\n5. Tuple data: {person}, Name: {person[0]}")
# person[1] = 26 # Uncommenting this line will cause a TypeError (immutability)
```

Results

Shows intermediate and final lists, statistics, tuple access, and the result of the list comprehensions.

Analysis & Inference

The list mutability methods show that lists are modified in place, which can save memory but requires careful state management. The deduplication process is an excellent example

Table 8: List and Tuple Operation Results

Operation	Initial	Final/Result	Feature Demonstrated
Mutability	`[5, 2, 9, 2, 7, 5, 1, 6]`	`[99, 9, 7, 5, 1, 6, 10]`	`append, insert, remove, pop`
Deduplication	`[... 9, 7, 5, 1, 6, 10]`	`[99, 9, 7, 5, 1, 6, 10]`	`set` used for efficient unique check
Comprehension	Unique list	`[9801, 81, 49, 25, 1, 36, 100]`	Concise list generation (Map)
Tuple Access	`("Alice", 25, "NY")`	Alice	Immutable record data

of using a `set` (fast lookups) and a `list` (order preservation) together. List comprehensions are confirmed to be highly **Pythonic**—readable and efficient—for generating new lists from existing data. Tuples enforce data integrity by preventing accidental modifications, ideal for fixed data records.

Conclusion

Lists and tuples are correctly used to manage ordered collections. List mutability, tuple immutability, and the efficiency of list comprehensions are verified.

Experiment 8: Dictionaries and Sets (Mapping and Membership)

Aim

Master the use of **Dictionaries** for key-value storage (mapping) and **Sets** for managing unique elements and performing set-theoretic operations.

Theory

Dictionaries (`dict`): Dictionaries are dynamic collections that store data as `key: value` pairs. They are unordered (in Python versions before 3.7) or insertion-ordered (Python 3.7 |).

- **Keys:** Must be unique and **hashable** (immutable types like strings, numbers, or tuples).
- **Efficiency:** Dictionary lookups, insertions, and deletions have an average time complexity of $O(1)$, making them extremely efficient for mapping data.
- **Safe Access:** `dict.get(key, default)` is the preferred method for key access, as it returns a specified `default` value (often `None` or 0) instead of raising a `KeyError` if the key is not found.

Sets (`set`): Sets are unordered collections of unique, hashable elements, defined by curly braces (`{}`) or the `set()` constructor.

- **Membership Testing:** Because sets are implemented using hash tables, checking for the existence of an element (`element in set`) is highly efficient ($O(1)$ average time).
- **Set Algebra:** Sets simplify algebraic operations on groups of data:
  - **Union (`A | B`):** All elements in A or B.
  - **Intersection (`A & B`):** Elements common to both A and B.
  - **Difference (`A - B`):** Elements in A but not in B.

Design & Implementation

Build a script that demonstrates both data structures:

1. **Word Frequency Counter:** Read a sentence and use a dictionary with `dict.get()` to count word occurrences.
2. **Interactive Contact Book:** Use a dictionary to implement a menu-driven program supporting `Add`, `Lookup`, and `Delete` by name.
3. **Set Operations:** Define two sets and calculate their union, intersection, and differences using operator syntax.
4. **New Task:** Dictionary Comprehension - Create a dictionary mapping key strings to their lengths using the concise comprehension syntax: `{k: v for ...}`

Code (Python)

```python
import string

# 1. Word Frequency Counter
text = "The quick brown fox jumps over the lazy brown dog"
freq = {}
for w in text.split():
        w = w.strip(string.punctuation).lower() # Clean and normalize the word
        freq[w] = freq.get(w, 0) + 1 # Safe increment using dict.get
print(f"Word Frequency: {freq}")

# 2. Interactive Contact Book
book = {"Alice": "555-0101", "Bob": "555-0202"}
print("\n--- Contact Book (Interactive) ---")
while True:
cmd = input("(A)dd (L)ookup (D)elete (Q)uit: ").strip().upper()
if cmd == "A":
        name = input("Name: "); phone = input("Phone: ")
        book[name] = phone
elif cmd == "L":
        name = input("Name to lookup: ")
        # Safe lookup: provides "not found" if key is absent
        print(f"Phone for {name}: {book.get(name, 'not found')}")
elif cmd == "D":
        name = input("Name to delete: ")
        if name in book:
                del book[name]
                print(f"{name} deleted.")
        else:
                print("Name not found.")
elif cmd == "Q":
        break
print(f"Current entries: {list(book.keys())}") # Show current keys

# 3. Set Operations
A = {1, 2, 3, 4, 5}
B = {4, 5, 6, 7, 8}
print("\n--- Set Operations ---")
print(f"Set A: {A}, Set B: {B}")
print(f"Union (A | B): {A | B}")
print(f"Intersection (A & B): {A & B}")
print(f"Difference (A - B): {A - B}")

# 4. Dictionary Comprehension
words = ["alpha", "beta", "gamma", "delta"]
word_lengths = {word: len(word) for word in words if len(word) > 4}
print(f"\nWord Lengths (>4): {word_lengths}")
```

Results

The execution confirms $O(1)$ lookups in the contact book and correct algebraic results from sets.

Table 9: Dictionary and Set Operation Verification

Structure/Task	Input/Initial	Observed Final State	Concept Verified
Dictionary (Freq.)	"brown fox brown dog"	`'brown': 2, 'fox': 1...`	`dict.get(key, 0) + 1` for counting
Contact Book (Lookup)	Lookup "Charlie"	Phone: not found	Safe access using `dict.get()`
Set Intersection	A={1..5}, B={4..8}	`{4, 5}`	Common elements identified efficiently
Dict Comp.	`"alpha", "beta"`	`'alpha': 5, 'delta': 5`	Concise mapping and filtering

Analysis & Inference

The use of `dict.get(w, 0)` in the frequency counter is a key Python idiom for initializing and incrementing counts without needing an explicit `if/else` check for key existence. The interactive contact book confirms the dictionary's role as a versatile, mutable map. Sets are confirmed to handle group algebra (`union, intersection`) efficiently and are excellent for quick membership tests due to their use of hashing.

Conclusion

Dictionaries and sets are implemented correctly for mapping and membership tasks. The ability to choose between these two structures based on the need for unique keys/elements versus ordered storage is established.

Experiment 9: Files and Exceptions (Text, CSV-like, JSON-like)

Aim

Implement file I/O for text data using the **context manager**; practice robust data parsing; and utilize **exception handling** (`try/except`) to manage runtime errors gracefully.

Theory

File I/O and Context Managers: The interaction with external files (Input/Output) is a critical component of persistence. The `open()` function returns a file object. The `with open(path, mode) as f:` statement is the preferred method, as it uses a **context manager**. This guarantees that the file resource is automatically and safely closed, even if errors occur during processing.

- **Modes:** `r` (read), `w` (write/overwrite), `a` (append).
- **Encoding:** Using `encoding="utf-8"` is standard practice for handling the widest range of characters.

Exception Handling: Program errors that occur during runtime are called **exceptions**. Robust programming requires anticipating and managing these errors using the `try/except` structure.

- `try`: The code block that may raise an exception.
- `except ExceptionType`: The block that executes if a specific exception (e.g., `ValueError`, `FileNotFoundError`) is raised in the `try` block. This allows the program to recover (e.g., skip a bad line) instead of crashing.

Data Serialization: Saving complex Python objects (like lists or dictionaries) to a text file requires serialization (converting the object to a string).

- `repr()`: Generates a string representation of an object that is valid Python syntax.
- `ast.literal_eval`: The built-in `ast` module's `literal_eval` function is the **safe** way to deserialize Python literal strings (containing lists, dictionaries, strings, numbers, etc.) back into their corresponding Python objects. It specifically rejects any string containing executable code, avoiding the severe security risks associated with the `eval()` function.

Design & Implementation

1. Write structured user data (Name, Age) to a `.txt` file using `w` mode, ensuring a header and a deliberately bad data line for testing.
2. Read the file back and use `try/except ValueError` inside the loop to calculate the average age while gracefully skipping malformed lines (e.g., non-numeric age).
3. Demonstrate file appending using `a` mode.
4. Store and retrieve a dictionary object using `repr()` for safe serialization and `ast.literal_eval` for safe deserialization.

Code (Python)

```python
import ast
import os
data_path = "people\_data.csv"
json\_path = "config\_data.txt"

# 1. Text Write using 'w' mode (overwrites) and header
print("--- Writing Initial Data (w mode) ---")
with open(data_path, "w", encoding="utf-8") as f:
f.write("Name,Age\n") # Header
f.write("Alice,30\n")
f.write("Bob,25\n")
f.write("Charlie,NA\n") # Intentional bad line for error testing

# 2. Append Data using 'a' mode
with open(data_path, "a", encoding="utf-8") as f:
f.write("David,40\n")
print(f"Data appended to {data_path}.")

# 3. CSV-like Parse with Try/Except
ages = []
print("\n--- Parsing Data with Error Handling ---")
try:
        with open(data_path, "r", encoding="utf-8") as f:
        next(f) # Skip header line
        for line in f:
                line = line.strip()
                if not line: continue

try:
        # Expecting format: Name,Age
        name, age_str = line.split(",")
        age = int(age_str)
        ages.append(age)
except ValueError:
        # Catches both split error (wrong format) and int
           conversion error
        print(f"[ERROR] Skipping invalid entry: {line}")

avg_age = sum(ages)/len(ages) if ages else 0
print(f"Successfully processed {len(ages)} valid entries.")
print(f"Average Age: {round(avg_age, 2)}")
except FileNotFoundError:
print(f"[FATAL ERROR] File {data_path} not found.")

# 4. JSON-like Safe Save/Load
data = {"course": "UCEST105", "count": len(ages), "ages": ages}
with open(json_path, "w", encoding="utf-8") as f:
        f.write(repr(data)) # repr() creates a valid Python string
            representation
```

```
48
49 print(f"\n--- Safe Data Loading ({json_path}) ---")
50 with open(json_path, "r", encoding="utf-8") as f:
51         content = f.read()
52 try:
53         loaded_data = ast.literal_eval(content)
54         print("Data loaded successfully.")
55         print(f"Loaded Course: {loaded_data.get('course')}")
56         except (ValueError, SyntaxError):
57         print("[ERROR] Failed to safely evaluate data. File
              content corrupted.")
```

Results

The results demonstrate both successful data flow and the necessary defensive programming techniques for file I/O.

Table 10: File I/O and Exception Handling Verification

File Mode/Operation	Key Action	Observed Console Output	Concept Verified
`w` and `a`	Write Alice, Append David	File contains Alice, Bob, David	File modes and `with open()`
`try/except`	Reading line `Charlie,NA`	`[ERROR] Skipping invalid entry: Charlie,NA`	Graceful handling of `ValueError`
CSV-like Parse	Valid lines	`Successfully processed 3 valid entries`	Data validation and conversion
Safe Load	Read `config_data.txt`	`Loaded Course: UCEST105`	`ast.literal_eval` safety

Analysis & Inference

The use of the `with open()` context manager ensures resource cleanup. The `try/except ValueError` block around the parsing step is critical; it allows the program to continue processing valid data even when encountering corrupted records, which is essential for robust data pipelines. `ast.literal_eval` is confirmed as the secure method for deserializing simple Python data structures from a text file, safeguarding against malicious code injection that could occur with `eval()`.

Conclusion

File read/write operations and mode differences (`w`, `a`) are established. Robust data parsing and graceful error recovery using `try/except` are correctly implemented, making the program resilient to common I/O failures.

Experiment 10: Simple Project with Modules and Classes

Aim

Integrate Python fundamentals by creating a small, menu-driven application ("Student Manager") that uses an Object-Oriented approach (`class`), organized **modules**, and file persistence.

Theory

Object-Oriented Programming (OOP) and Classes: OOP is a programming paradigm based on the concept of "objects," which are instances of **classes.** A class is a blueprint for creating objects, defining a set of **attributes** (data, variables) and **methods** (behavior, functions) that operate on that data.

- **Constructor (`__init__`):** The method automatically called when a new object is created. It is used to initialize the object's attributes.
- **String Representation (`__str__`):** A special method that returns a human-readable string representation of the object, used when the object is passed to `print()`.

Modularity: Breaking a large program into smaller, interconnected **modules** (`.py` files) is key to managing complexity. Each module should handle a single, distinct concern (e.g., `student.py` for data logic, `store.py` for persistence). This improves **code reusability** and **maintainability.**

Persistence and Integration: This experiment ties together I/O (Experiment 9) and OOP (Classes) by demonstrating how to save a list of custom objects to an external file and reconstruct them back into memory. This simple form of persistence is crucial for any stateful application.

Design & Implementation

1. `student.py`: Define the `Student` class with attributes (`name, roll, marks`) and methods (`percent, result, __str__`).
2. `store.py`: Implement the utility functions `save_students()` (saving to pipe-separated format) and `load_students()` (parsing the file back into `Student` objects). Include `try/except` for file loading robustness.
3. `main.py`: Create the interactive loop and menu (`Add, List, Save, Load, Quit`) that manages a master `list` of `Student` objects and orchestrates the calls to the `store` module.

Code (Python)

```python
# student.py
class Student:
        """Represents a student with roll, name, and marks.
           Calculates percentage and result."""
        def __init__(self, name, roll, marks):
        self.name = name
        self.roll = roll
        self.marks = marks  # List of integers

        def percent(self):
                # Prevent ZeroDivisionError if marks list is empty
                return sum(self.marks) / (len(self.marks) or 1)

        def result(self):
                return "PASS" if self.percent() >= 40 else "FAIL"

        def __str__(self):
                # Enhanced string representation for listing
                return f"Roll: {self.roll:<5} Name: {self.name
                   :<15} Pct:        {self.percent():.1f}% | {self
                   .result()}"
```

```python
# store.py
from student import Student
import os

def save_students(path, students):
        """Saves a list of Student objects to a pipe-separated
           text file."""
        try:
                with open(path, "w", encoding="utf-8") as f:
                        for s in students:
                                marks_str = ",".join(str(m) for m
                                   in s.marks)
                                f.write(f"{s.roll}|{s.name}|{
                                   marks_str}\n")
                return True
        except Exception as e:
                print(f"[SAVE ERROR] Failed to save file: {e}")
                return False

def load_students(path):
        """Loads a list of Student objects from a text file,
           handling errors."""
        students = []
        if not os.path.exists(path):
                print(f"File '{path}' not found. Starting fresh.")
        return students

        try:
```

```
                with open(path, "r", encoding="utf-8") as f:
                        for line in f:
                                line = line.strip()
                                if not line: continue

                roll, name, marks_str = line.split("|")
                # Robust parsing of marks
                marks = [int(x.strip()) for x in marks_str.split(",") if x.strip() and x.strip().isdigit()]
                students.append(Student(name, roll, marks))
                print(f"Successfully loaded {len(students)} students.")
        except Exception as e:
                print(f"[LOAD ERROR] File corruption or unexpected format: {e}. Data cleared.")
                return [] # Return empty list on severe corruption
        return students
```

```
# main.py
from student import Student
import store

STUDENT_FILE = "students.txt"
students = store.load_students(STUDENT_FILE)  # Load on startup


def add_student():
        name = input("Name: ")
        roll = input("Roll: ")
        marks_input = input("Marks (comma separated, e.g., 80,75,90): ")
        try:
                marks = [int(x.strip()) for x in marks_input.split(",") if x.strip()]
                if not marks:
                        raise ValueError("No valid marks entered.")
                        students.append(Student(name, roll, marks))
                print("Student added successfully.")
        except ValueError as e:
                print(f"[INPUT ERROR] Invalid marks format. {e}")


def list_students():
        print("\n--- Student Records ---")
        if not students:
                print("No students recorded.")
                return
        for s in students:
                print(s)
```

```
        print("-----------------------")


# Main menu loop
while True:
        print("\n(1) Add (2) List (3) Save (4) Load (5) Quit")
        ch = input("Choice: ").strip()
        if ch == "1":
                add_student()
        elif ch == "2":
                list_students()
        elif ch == "3":
                store.save_students(STUDENT_FILE, students)
        elif ch == "4":
                students = store.load_students(STUDENT_FILE)
        elif ch == "5":
                break
        else:
                print("Invalid choice. Please select 1-5.")
```

Results

Users can add students, list them with percentages and pass/fail, and persist to a text file.

Table 11: Project Integration and Output Verification

Action (Menu)	Input Example	Core Functionality	Concepts Integrated
Add (1)	Roll: 101, Marks: 30, 40	New `Student` object created	OOP Class `__init__`, List
List (2)	(Internal data)	`...Pct: 35.0% | FAIL`	`__str__` method, `percent()` logic
Save (3)	(Internal data)	`students.txt` created/overwritten	Modularity (`store.py`), File I/O
Load (4)	Non-existent file	`File 'students.txt' not found...`	Exception Handling (`FileNotFoundError`)

Analysis & Inference

The project structure is a complete example of a simple Python application, demonstrating **separation of concerns**: `student.py` handles data logic, `store.py` handles persistence, and `main.py` handles the user interface. Using the `Student` **class** ensures data integrity and consistency, as all student-related calculations (percentage, result) are encapsulated within the object itself. The robust loading function in `store.py` (using `os.path.exists` and `try/except`) is essential for professional applications.

Conclusion

A fully integrated, menu-driven Python project was successfully implemented. The experiment confirms mastery of OOP, modular design, structured I/O, and error handling, achieving the final project objective.