COMP 1023 Midterm Exam - Spring 2026 - HKUST

Date: April 11, 2026 (Saturday)
Time Allowed: 2 hours, 2:00–4:00 pm
Instructions:

1. This is a closed-book, closed-notes examination.
2. There are **8** questions on **17** pages (including this cover page, appendix and 4 blank pages at the end). You may tear off the appendix and the blank pages at the back.
3. Write your answers in the space provided in the answer book using black/blue ink. *NO pencils please, otherwise you are not allowed to appeal for any grading disagreements.*
4. All programming code in your answers must be written in Python version as taught in class.
5. Unless otherwise stated, you are **NOT** allowed to define additional classes, helper functions (including inner functions) and use global variables, nor any library functions not mentioned in the questions.
6. **Type hinting does not need to be included in your code.**
7. No electronic devices are allowed, except for an HKEAA approved calculator.

Student Name	SOLUTION & MARKING SCHEMES		
Student ID			
Venue		Seat Number	

Total (For T.A. Use Only)

For T.A. Use Only

Problem	Topic	Score		Total	Full Mark
1	True/False Questions				10
2	Python Operations				7
3	Branching	(a)			4
		(b)			
4	Looping	(a)			9
		(b)			
5	Lists	(a)			16
		(b)			
		(c)			
6	Functions	(a)			16
		(b)			
		(c)			
		(d)			
7	Fare Helper	(a)			22
		(b)(i)			
		(b)(ii)			
		(c)			
		(d)(i)			
		(d)(ii)			
		(d)(iii)			
8	Strings and Lists Utilities	(a)(i)			16
		(a)(ii)			
		(a)(iii)			
		(b)(i)			
		(b)(ii)			

Problem 1 [10 points] True/False Questions

Solution:

Question	(a)	(b)	(c)	(d)	(e)	(f)	(g)	(h)	(i)	(j)
Answer	F	F	T	T	T	F	F	T	F	F

Scheme:

- 1 point for each correct answer, adding up to a total of 10 points.

Problem 2 [7 points] Python Operations

Solution:

After finishing the statement on line	`x`	`y`	`z`	`check`
5	4.0	5.0	/	/
7	4.0	15.0	/	/
9	4.0	15.0	2	/
11	4.0	15.0	2	True

Marking Scheme:

- 0.5 point for each correct non-empty table entry, 5.5 points in total.
- 0.3 point for each correct empty table entry, 1.5 points in total.
  No point is given if they write "None" instead of '/', or they don't write '/'. If they write "Not been created", -0.15 points each.
- Note:
  - If students enter '/' in the cells under 4.0 or leave them blank, consider it as if they have entered the correct value of 4.0.
  - Likewise, if they enter '/' in the cells under 15.0 or leave them blank, treat it as if they have entered the correct value of 15.0.
  - Similarly, if they enter '/' in the cell under 2 or leave it blank, consider it as if they have entered the correct value of 2.
  - If students write 'ínstead of '/', no penalty.
  - If students leave a column all blank, no point is given.
  - If students write "true" instead of "True", no penalty. If they write "T", no penalty.
  - If students write 4 instead of 4.0, 15 instead of 15.0, 2.0 instead of 2, -0.25 points each.
  - If students write "1" instead of "True", -0.25 points.

Problem 3 [4 points] Branching

Solution:

(a) McDonald's

(b) 1023
1023
nothing

Marking Scheme:

(a) 1 point for the correct answer.

(b) 1 point for each correct line. 3 points in total.

Problem 4 [9 points] Looping

Solution:

(a)

Iteration	`i`	`val`	`total`
1	0	3	0
2	1	4	3
3	2	-2	-1 (unchanged)
4	4	1	-1
5	5	0	0 (loop breaks)
6			
7			

(b) `0`

Marking Scheme:

(a) 0.5 points for each correct value. 7.5 points in total. 0.5 point for leaving all the remaining cells empty.

(b) 1 point for the correct final return value. (Total 1 point)

Problem 5 [16 points] Lists

Solution:

(a)
```
def contains_all(lst: list[int]) -> bool:
    for x in range(1, 10):                  # 1 point for the loop condition
        if x not in lst:                    # 1.5 points for this case
            return False
    return True                             # 0.5 points for this case
```

(b)
```
def check_rows_and_columns(grid: list[list[int]]) -> bool:
    for i in range(9):                      # 1 point for the loop condition
        if not contains_all(grid[i]):       # 1.5 points for this case
            return False
        if not contains_all([grid[j][i] for j in range(9)]): # 3 points for this case
            return False
    return True                             # 0.5 point for this case
```

(c)
```
def check_boxes(grid: list[list[int]]) -> bool:
    for r1 in range(3):                             # 1 point for the outer loop
        for c1 in range(3):                         # 1 point for the inner loop
            lst = [grid[r1 * 3 + r2][c1 * 3 + c2]   # 3 points
                       for r2 in range(3) for c2 in range(3)]
            if not contains_all(lst):               # 1.5 points for this case
                return False
    return True                                     # 0.5 point for this case
```

Marking Scheme:

- See Solution.
- Deduct 0.25 pts for each syntax error. At most 1 point for each part.

Problem 6 [16 points] Functions

Solution:

(a)
```
def get_grade(score: int) -> str:
    if score >= 90:
        return "A"
    elif score >= 80:
        return "B"
    elif score >= 70:
        return "C"
    elif score >= 60:
        return "D"
    else:
        return "F"
```

(b)
```
def count_grades(scores: list[int]) -> list[int]:
    counts: list[int] = [0, 0, 0, 0, 0]
    for s in scores:
        grade: str = get_grade(s)
        if grade == "A":
            counts[0] += 1
        elif grade == "B":
            counts[1] += 1
        elif grade == "C":
            counts[2] += 1
        elif grade == "D":
            counts[3] += 1
        else:
            counts[4] += 1
    return counts
```

(c)
```
def apply_bonus(scores: list[int], bonus: int = 5) -> list[int]:
    adjusted: list[int] = []
    for s in scores:
        new_score: int = s + bonus
        if new_score > 100:
            new_score = 100
        adjusted.append(new_score)
    return adjusted
```

(d)
```
def find_rank(scores: list[int], student_index: int) -> int:
    student_score: int = scores[student_index]
    rank: int = 1
    for s in scores:
        if s > student_score:
            rank += 1
    return rank
```

Marking Scheme:

(a) **get_grade (4 points)**

- 0.5 pts: correctly defining the function with name `get_grade` and parameter `score`. Not allowed to define parameter score as a list.
- 0.5 pts: correctly returning `"A"` when `score >= 90`.
- 1 pt: correctly returning `"B"` when `80 <= score < 90`.
- 1 pt: correctly returning `"C"` when `70 <= score < 80`.
- 0.5 pts: correctly returning `"D"` when `60 <= score < 70`.
- 0.5 pts: correctly returning `"F"` otherwise.
- Deduct 0.5 pts if boundary conditions are incorrect (e.g., using `>` instead of `>=`).
- If there is a return issue, but the logical comparison is correct, deduct 1 point.
- If there is a Name Error (e.g., caused by missing the `" "` for the strings) while returning the strings, deduct 1 point. For cases that you only have one name error, deduct 0.25 points.
- If there is a loop inside the function (except for those using it properly, i.e., aligns with the function's purpose), deduct 0.5 points. Same for list operations.
- If there is an extra statement that affects the function to run properly, deduct 0.5 points.

(b) **count_grades (3 points)**

- 0.5 pts: correctly defining the function with name `count_grades` and parameter `scores`.
- 0.5 pts: initializing a list of 5 zeros.
- 0.5 pts: using a for-loop to iterate through the scores.
- 1 pt: calling `get_grade` from part (b) to determine each grade. 0 points for this criterion if `get_grade` is not reused.
- 0.5 pts: correctly incrementing the appropriate count and returning the list.

(c) **apply_bonus (4 points)**

- 0.5 pts: correctly defining the function with name `apply_bonus`.
- 0.5 pts: correctly defining parameters `scores` and `bonus`. 0 point for this part if the parameter is not strictly named `bonus`. The problem description explicitly specifies and holds the parameter name `bonus`, requiring strict naming compliance.
- 1 pt: correctly using a default parameter value of `5` for `bonus`.
- 0.5 pts: creating and returning a **new** list (not modifying the original). 0 point for this part if the original list is modified in-place.
- 0.5 pts: correctly adding `bonus` to each score using a for-loop. Hard coding the loop range (e.g., using range(10)) is not allowed.
- 1 pt: correctly capping scores at 100 (using branching). Using the `min()` function to cap the score is also allowed as a valid alternative to branching.

(d) **find_rank (5 points)**

- 0.5 pts: correctly defining the function with name `find_rank` and parameters `scores` and `student_index`.
- 1 pt: correctly retrieving the student's score using `scores[student_index]`.
- 3 pts: correct storing logic. -1 pt if the essence of the sorting logic is correct, but there is a mistake, e.g., reversed sorting, wrong usage of sorting function, out of the bounds error because a "-1" was forgotten, etc.
- 0.5 pts: correctly returning the rank.

General deductions:

- Deduct 0.25 pts for each syntax error. At most 1 point for each part.

Problem 7 [22 points] Fare Helper

Solution:

```
# Total points: 22

# ======================================================================
# PART (a) - Find fare for a specific station and class
# ======================================================================
def part_a(stations, classes, prices, input_station, input_class):
    """
    Find the fare for a given station and class.

    Args:
        stations: List of station names
        classes: List of class names
        prices: 2D list of prices
        input_station: Name of the station to lookup
        input_class: Name of the class to lookup

    Returns:
        The fare price as an integer
    """
    station_index = stations.index(input_station)
    class_index = classes.index(input_class)
    fare = prices[station_index][class_index]
    # ------------------------------------------------------
    # @GRADING Part (a) - 2 Points Total
    # [+0.5] Correctly finding the index of the station
    # [+0.5] Correctly finding the index of the class
    # [+1] Correctly looking up the fare in the 2D prices list
    #
    # Alternatives:
    # fare = prices[stations.index(input_station)][classes.index(input_class)] -> Full
    ↪  Marks
    # ------------------------------------------------------
    return fare


# ======================================================================
# PART (b) - Add new station and prices
# ======================================================================
def part_b(stations, prices, new_station, new_price):
    """
    Add a new station and its prices to the data.
```

```
    Args:
        stations: List of station names
        prices: 2D list of prices
        new_station: Name of the new station
        new_price: Comma-separated string of prices for the new station
    """
    stations.append(new_station)
    # ----------------------------------------------------------
    # @GRADING Part (b)(i) - 1 Point Total
    # [+1] Correctly appending new_station to the stations list
    #
    # Penalties:
    # [0] No partial marks for this part
    # ----------------------------------------------------------

    prices.append([int(p) for p in new_price.split(",")])
    # ----------------------------------------------------------
    # @GRADING Part (b)(ii) - 3 Points Total
    # [+1] Using .split(",") to separate the input string
    # [+1] Iterating over the split price strings and converting them to integers
    # [+1] Appending the new integer list as a row to the prices table
    #
    # Alternatives:
    # prices.append(list(map(int, new_price.split(",")))) -> Full Marks
    # ----------------------------------------------------------


# ======================================================================
# PART (c) - Find valid stations (strictly increasing prices)
# ======================================================================
def part_c(stations, prices):
    """
    Find stations where prices are strictly increasing across classes.

    Args:
        stations: List of station names
        prices: 2D list of prices

    Returns:
        List of valid station names
    """
    valid_stations = []
    for i, price in enumerate(prices):
        for j in range(len(price) - 1):
            if price[j] >= price[j+1]:
                break
```

```
    else:
        valid_stations.append(stations[i])
# ------------------------------------------------------
# @GRADING Part (c) - 6 Points Total
# [+1] Iterating over each fare row with its index (e.g., enumerate)
# [+1] Iterating over adjacent fare classes (e.g., range(len(price) - 1))
# [+1] Correctly checking if fare is NOT strictly increasing (>=)
# [+1] Using 'break' to exit the inner loop when a violation is found
# [+1] Using 'else' clause on the inner loop to detect no violations
# [+1] Appending the valid station to valid_stations
#
# Alternatives:
# --- Alternative 1: using enumerate(stations) ---
# valid_stations = []
# for i, station in enumerate(stations):
#     for j in range(len(prices[i]) - 1):
#         if prices[i][j] >= prices[i][j+1]:
#             break
#     else:
#         valid_stations.append(station)
#
# --- Alternative 2: using zip(stations, prices) ---
# valid_stations = []
# for station, price in zip(stations, prices):
#     for j in range(len(price) - 1):
#         if price[j] >= price[j+1]:
#             break
#     else:
#         valid_stations.append(station)
#
# --- Alternative 3: using range(len(stations)) ---
# valid_stations = []
# for i in range(len(stations)):
#     for j in range(len(prices[i]) - 1):
#         if prices[i][j] >= prices[i][j+1]:
#             break
#     else:
#         valid_stations.append(stations[i])
#
# Penalties:
# [-0.5] Improper usage of list.index()
# [-1] Improper usage of pop() and remove()
# No partial mark will be given if hard coding the part for comparing prices
# [-0.5] for hard coding the loop range
# ------------------------------------------------------
return valid_stations
```

```python
# ======================================================================
# PART (d) - Calculate average fares, promo stations, and min multiplier
# ======================================================================
def part_d(stations, prices):
    """
    Calculate average fares, find promo stations (avg < 200),
    and find the minimum price multiplier.

    Args:
        stations: List of station names
        prices: 2D list of prices

    Returns:
        Tuple of (avg_fare, promo_stations, min_multiplier)
    """
    # Part (d)(i) - Calculate average fare for each station
    avg_fare = [sum(row) / len(row) for row in prices]
    # ----------------------------------------------------------
    # @GRADING Part (d)(i) - 3 Points Total
    # [+1] Iterating over the rows of the fare table
    # [+1] Getting the sum of the fares for each station (sum(row))
    # [+1] Dividing by the number of classes (len(row)) to get the average
    # ----------------------------------------------------------

    # Part (d)(ii) - Find stations with average fare < 200
    promo_stations = [stations[i] for i in range(len(stations)) if avg_fare[i] < 200]
    # ----------------------------------------------------------
    # @GRADING Part (d)(ii) - 3 Points Total
    # [+1] Iterating over the station indices (e.g., range(len(stations)))
    # [+1] Checking the condition (avg_fare[i] < 200)
    # [+1] Extracting the correct station name (stations[i])
    #
    # Alternatives:
    # [station for i, station in enumerate(stations) if avg_fare[i] < 200] -> Full
    ↪  Marks
    # [s for s, a in zip(stations, avg_fare) if a < 200] -> Full Marks
    # ----------------------------------------------------------

    # Part (d)(iii) - Find minimum multiplier (Business / Second class ratio)
    min_multiplier = min(row[-1] / row[0] for row in prices)
    # ----------------------------------------------------------
    # @GRADING Part (d)(iii) - 4 Points Total
    # [+1] Using a generator expression / list comprehension to iterate over the rows
    ↪  of the fare table
```

```
    # [+1] Accessing the most luxurious class price (row[-1])
    # [+1] Calculating the ratio with the least luxurious class price (/ row[0])
    # [+1] Correctly applying min() to find the smallest multiplier
    # ------------------------------------------------------------

    return avg_fare, promo_stations, min_multiplier


# ======================================================================
# MAIN PROGRAM
# ======================================================================
if __name__ == "__main__":
    # Initialize data
    stations = ["Futian"]
    classes = ["Second", "First", "Premium", "Business"]
    prices = [[ 78, 125, 140, 234]]

    # Get user inputs
    input_station = input("Enter the station name: ")
    input_class = input("Enter the class name: ")

    fare = part_a(stations, classes, prices, input_station, input_class) # Call part
    ↪  (a)
    print(f"(a) fare: {fare}")

    new_station = input("Enter new station name: ")
    new_price = input(f"Enter prices for {new_station} (comma-separated): ")

    part_b(stations, prices, new_station, new_price) # Call part (b)
    print(f"(b)(i) stations: {stations}")
    print(f"(b)(ii) prices: {prices}")

    valid_stations = part_c(stations, prices) # Call part (c)
    print(f"(c) valid_stations: {valid_stations}")

    avg_fare, promo_stations, min_multiplier = part_d(stations, prices) # Call part
    ↪  (d)
    print(f"(d)(i) avg_fare: {avg_fare}")
    print(f"(d)(ii) promo_stations: {promo_stations}")
    print(f"(d)(iii) min_multiplier: {min_multiplier:.2f}")
```

Marking Scheme:

- See Solution.
- Deduct 0.25 pts for each syntax error. At most 1 point for each part.

Problem 8 [16 points] Strings and Lists Utilities

Solution:

(a) (i)
```
from typing import Any
def join(target: list[Any], sep: str) -> str:
    s = ""
    for i in target:
        s += str(i) + sep
    for _ in target:
        s = s[:-1]
    return s
```

(ii)
```
from typing import Any
def get_diagonal(target: list[list[Any]]) -> list[Any]:
    row_num = len(target)
    for i in range(row_num):
        if len(target[i]) != row_num:
            return []
    return [target[i][i] for i in range(row_num)]
#   return [row[i] for i, row in enumerate(target)]
```

(iii)
```
from typing import Any
def diag_to_str(target: list[list[Any]]) -> str:
    return join(get_diagonal(target), ",")
```

(b) (i)
```
from typing import Any
def mask_diag(target: list[list[Any]], val: Any) -> list[list[Any]]:
    return [
        [target[i][j] if i <= j else val for j in range(len(target))]
        for i in range(len(target))
    ]
```

(ii)
```
from typing import Any
def skip_list(target: list[Any], factor: int) -> list[Any]:
    return [target[i] for i in range(len(target)) if (i + 1) % factor] + \
        target[factor-1::factor]
```

Marking Scheme:

(a) For each subsubquestion, if the student acquired a score s, the score they will acquire per subsubquestion is $\max(s, 0)$.

(i) Before grading:

- NO points given if any non-`str()` function calls exist in the answer.
- 0.5 points deducted if the `return` keyword/statement is missing.
- 0.5 points deducted if they directly write `target[0]`, regardless of the possibility of empty target.

Then,

- 0.5 points for iterating through `target` (via index or direct iteration)
- 0.5 points for converting `i` to a string
- 1 points for adding `str(i)` and `sep` in the loop. If missing `str(i)`, e.g., `i + sep`, 0.5 points are given.
- 0.5 points for attempting to removing the extra separator from the string via indexing
- 0.5 points for correct calculation of the ending index `-len(sep)`.
- 0.5 points are DEDUCTED if the initial empty string is NOT initialized.
- 0.5 points are DEDUCTED if the case where the target is empty is handled incorrectly, but all the remaining are correct.

(ii) Before grading:

- 0.5 points deducted if the `return` keyword/statement is missing.

Then,

- Check whether the input is a square matrix (1.5 points)
  - 1 point for using an appropriate for loop to check whether the input is square.
  - 0.5 points for the return statement, awarded if the student writes return [] when the input is not square.
- Extract the diagonal entries (1.5 points)
  - 1 point for using appropriate for loop to extract the diagonal entries.
  - 0.5 point for the return statement, awarded if the student returns a resulting list, regardless of whether that list is completely correct.
- Any logic error or variable naming error will result in a deduction of 0.5 points.
- For example, using list as a variable name instead of the required target should incur a 0.5 points deduction.

(iii) Before grading:

- NO points given if more than one comprehension exist in the answer.
- NO points given if at least one semi-colon exist in the answer.
- 0.5 points deducted if the `return` keyword/statement is missing.

Then,

- 0.5 points for using a comma as the separator.
  - No points deducted if there are leading or trailing spaces in the string.
- 1 point for `get_diagonal(target)`.
- 0.5 points for passing the result of `get_diagonal(target)` and a comma into `join`.

(b) For each subsubquestion, if the student acquired a score s, the score they will acquire per subsubquestion is $\max(s, 0)$.

(i) Before grading:

- NO points given if more than one statement exist in the answer.
- NO points given if at least one semi-colon exist in the answer.
- 0.5 points deducted if the `return` keyword/statement is missing.

Then,

- 0.5 points for nested comprehensions of the indices of the 2 axes of `target` (NO points for direct iteration)
- 1 points for the comparison `i <= j` (index of row $\leq$ index of column)
- 0.5 points for using `target[i][j]` if the condition is `True`.
- 1 points for using `val` if the condition is `False`.

(ii) Before grading:

- NO points given if more than one statement exist in the answer.
- NO points given if at least one semi-colon exist in the answer.
- NO points given if more than one comprehension statement exist in the answer.
- 0.5 points deducted if the `return` keyword/statement is missing.

Then,

- 1 point for checking if `(i + 1) % factor` is not `False`.
- 0.5 points for acquiring the relevant elements of `target` based on the condition above.
- 0.5 points for concatenating two lists with `+` operator.
- 0.5 points for the slicing starting index of `factor - 1`.
- 0.5 points for the empty slicing ending index (leaving it as default) or a value of `len(factor)`.
- 0.5 points for the slicing step of `factor`.
- 0.5 points for slicing `target` with a slice

---------- END OF PAPER ----------