COMP 1023 Midterm Exam - Spring 2026 - HKUST
Question Book

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 **22** 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			
Student ID			
Venue		Seat Number	

This page is intentionally left blank.

Problem 1 [10 points] True/False Questions

Indicate whether the following statements are true or false by putting **T** or **F** in the given table in the answer book. You get 1 point for each correct answer.

(a) The output of the following program is `200, 200`.

```
def main() -> None:
    x = y = 100
    x = 200
    print(f"{x}, {y}")

if __name__ == "__main__":
    main()
```

(b) In Python, the following two code snippets always produce the same output for any integer value of x:

Snippet A:

```
def main():
    x = int(input())
    if x > 0:
        print("A")
    elif x > -5:
        print("B")

if __name__ == "__main__":
    main()
```

Snippet B:

```
def main():
    x = int(input())
    if x > 0:
        print("A")
    if x > -5:
        print("B")

if __name__ == "__main__":
    main()
```

(c) The output of the following program is `COMP1023`.

```
def main() -> None:
    x: bool = bool("False")
    if x:
        print("COMP", end="")
    if x == True:
        print("1023")

if __name__ == "__main__":
    main()
```

(d) The output of the following program is `0 1 2 3 .`

```
def main() -> None:
    for i in range(4):
        print(i, end=" ")
        i += 2

if __name__ == "__main__":
    main()
```

(e) The output of the following program is `[10, 6]`.

```
def main() -> None:
    values: list[int] = [2, 4, 6, 8, 10]
    print(values[:1:-2])

if __name__ == "__main__":
    main()
```

(f) The output of the following program is `5`.

```
def main() -> None:
    lst: list[int] = [1, 2, 3, 4, 5]
    lst[1:4] = [10]
    print(len(lst))

if __name__ == "__main__":
    main()
```

(g) The output of the following program is `[1, 2, 3, 4]`.

```
from typing import Any

def process(values: list[Any]) -> list[Any] | None:
    result: list[Any] = []
    for v in values:
        if v:
            result.append(v)
        else:
            return result
    return None

def main() -> None:
    data: list[int] = [1, 2, 0, 3, 4]
    print(process(data))

if __name__ == "__main__":
    main()
```

(h) The following program will raise an error.

```
def main():
    x: list[int] = [1023]
    x = x * 4

    for i in range(4):
        print(list([x])[i], end=' ')

if __name__ == "__main__":
    main()
```

(i) The output of the following program is `[[99, 2], [1, 2]]`.

```
def main() -> None:
    row: list[int] = [1, 2]
    grid: list[list[int]] = [row, row]
    grid[0][0] = 99
    print(grid)

if __name__ == "__main__":
    main()
```

(j) The following program will result in an infinite loop.

```
import random

def main() -> None:
    my_list: list[int] = list(range(10, 0, -1))
    random.shuffle(my_list)
    x: int = 0
    while True:
        x = my_list[x]

if __name__ == "__main__":
    main()
```

Problem 2 [7 points] Python Operations

Analyze the program below, which uses arithmetic operations, simultaneous assignment, augmented assignment, comparison operations, and logical operations.

```
1  def main():
2      x: int = 11
3      y: float = 2.0
4
5      x, y = y + x // 4, x % 4 + y
6
7      y *= x - 1
8
9      z: int = 2 ** 3 ** 1 // 4
10
11     check: bool = y >= 15 and not (z != 2)
12
13 if __name__ == "__main__":
14     main()
```

Complete the given table in the answer book. After finishing the statement on each numbered line, write down the values of `x`, `y`, `z`, and `check`. If a variable has not been created yet, write `/` in the table cell. Also, it is assumed that no floating-point precision error will be encountered.

Problem 3 [4 points] Branching

(a) [1 point] What is the output of the following program?

```
def main():
    lunch_type: str = "fast food"
    print("The lunch should be: " + "salad" \
          if lunch_type == "healthy food" else "McDonald's")

if __name__ == "__main__":
    main()
```

(b) [3 points] What is the output of the following program?

```
def main():
    lunch_type: int = 1021
    _: int = 1023

    print(_)

    match lunch_type:
        case 1:
            print("salad")
        case 2:
            print("McDonald's")
        case _:
            print(_)
            _ = "nothing"

    print(_)

if __name__ == "__main__":
    main()
```

Problem 4 [9 points] Looping

Analyze the function `process_signals(signals)` below. It uses a `while` loop to process a list of integers and calculates a total value.

```
1  def process_signals(signals: list[int]) -> int:
2      total: int = 0
3      i: int = 0
4
5      while i < len(signals):
6          val: int = signals[i]
7
8          # *** Value of i, val, and total at this point ***
9
10         if val == 0:
11             break
12
13         if val < 0:
14             i += (-val)
15             continue
16
17         if val % 2 != 0:
18             total += val
19         else:
20             total -= val
21
22         i += 1
23
24     return total
```

(a) [8 points] Given the list `data = [3, 4, -2, 8, 1, 0, 5]`, when we call the function `process_signals(data)`, please trace the values of `i`, `val`, and `total` at **line 8**, specifically at the point marked with comment:
`# *** Value of i, val, and total at this point ***`
for each iteration. Complete the tracing table in your answer book. Stop when the loop terminates, and leave the remaining table row(s) blank.

(b) [1 point] What is the final return value of `process_signals(data)`?

Problem 5 [16 points] Lists

In this question, you will implement a solution checker for Sudoku. Sudoku is a game where the player will fill out a 9×9 grid with numbers according to some rules (See Figure 1).

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

Figure 1: An example Sudoku game

The rules of Sudoku are as follows:

1. Each 3×3 grid highlighted in bold must contain the numbers from 1 to 9.
2. Each row must contain the numbers from 1 to 9.
3. Each column must contain the numbers from 1 to 9.

In this question, the grid will be represented by a 2-dimensional list `grid`, where `grid[x][y]` will store the number of the grid at the `x`-th row and the `y`-th column. Assume the dimensions of the grid is 9×9. Also, **at most 7 statements** are allowed for each question. A single list comprehension counts as a single statement.

(a) [3 points] Given the function header below, implement `contains_all` that checks if the input list contains all numbers from 1 to 9. Assume the list `lst` only has 9 elements.

```
def contains_all(lst: list[int]) -> bool:
```

(b) [6 points] Given the function header below, implement `check_rows_and_columns` that checks that all rows and columns satisfy the requirement.

```
def check_rows_and_columns(grid: list[list[int]]) -> bool:
```

(c) [7 points] Given the function header below, implement `check_boxes` that checks that all 3×3 grid satisfy the requirement.

```
def check_boxes(grid: list[list[int]]) -> bool:
```

Problem 6 [16 points] Functions

In this question, you will create several Python functions to analyze a list of student exam scores. The main program has already been written for you as follows:

```
def main() -> None:
    scores: list[int] = [85, 92, 78, 95, 60, 88, 73, 100, 67, 55]

    # Part (a)
    print("Grade for 85:", get_grade(85))
    print("Grade for 55:", get_grade(55))

    # Part (b)
    counts: list[int] = count_grades(scores)
    print("Grade counts [A, B, C, D, F]:", counts)

    # Part (c)
    adjusted_scores: list[int] = apply_bonus(scores)
    print("Adjusted scores:", adjusted_scores)
    adjusted_scores2: list[int] = apply_bonus(scores, 10)
    print("Adjusted scores (+10):", adjusted_scores2)

    # Part (d)
    print("Rank of student 0:", find_rank(scores, 0))
    print("Rank of student 3:", find_rank(scores, 3))
    print("Rank of student 7:", find_rank(scores, 7))

if __name__ == "__main__":
    main()
```

When all functions are correctly implemented, the output should look like this:

```
Grade for 85: B
Grade for 55: F
Grade counts [A, B, C, D, F]: [3, 2, 2, 2, 1]
Adjusted scores: [90, 97, 83, 100, 65, 93, 78, 100, 72, 60]
Adjusted scores (+10): [95, 100, 88, 100, 70, 98, 83, 100, 77, 65]
Rank of student 0: 5
Rank of student 3: 2
Rank of student 7: 1
```

Implement the following functions so that the program runs correctly and produces the expected output. **For each function, you need to write the complete function header and the function body.** Type hinting does not need to be included in your code. You may assume that all score lists are non-empty and all scores are integers between 0 and 100 inclusive.

(a) [4 points] Implement the function `get_grade` that takes an integer score and returns the corresponding letter grade as a string, according to the following table:

Score Range	Grade
90 – 100	`"A"`
80 – 89	`"B"`
70 – 79	`"C"`
60 – 69	`"D"`
0 – 59	`"F"`

(b) [3 points] Implement the function `count_grades` that takes a list of integer scores and returns a list of 5 integers representing the count of each grade in the order `[count_A, count_B, count_C, count_D, count_F]`. You **must** reuse the `get_grade` function from part (a).

(c) [4 points] Implement the function `apply_bonus` that takes a list of integer scores and an optional integer parameter `bonus` with a default value of `5`. It returns a **new** list where each score is increased by `bonus`, but no score may exceed 100.

(d) [5 points] Implement the function `find_rank` that takes a list of integer scores and an integer `student_index`. It returns the **rank** (as an integer) of the student at the given index, where rank 1 means the highest score. If multiple students share the same score, they receive the same rank. For example, if the scores are `[70, 90, 90, 80]`, the student at index 0 (score 70) has rank 4, the students at indices 1 and 2 (score 90) both have rank 1, and the student at index 3 (score 80) has rank 3.

Problem 7 [22 points] Fare Helper

站點 Station	成人票 Adult Ticket* 車廂等級 Class of Travel			
	二等 Second	一等 First	特等 Premium	商務 Business
短途列車 Short-haul train				
福田 Futian	$78	$125	$140	$234
深圳北 Shenzhenbei	$86	$138	$156	$260
光明城 Guangmingcheng	$109	$175	$196	$326
虎門 Humen	$204	$308	$349	$428
慶盛 Qingsheng	$212	$319	$363	$447
廣州南 Guangzhounan	$247	$371	$423	$519

Figure 2: High-speed Rail Fare Table

Figure 2 shows a fare table for the Guangzhou–Shenzhen–Hong Kong high-speed rail. All prices listed are ticket prices from Hong Kong West Kowloon Station to various Mainland destinations, with different columns showing the prices for different *classes of travel*. In this problem, you will write a Python "fare helper" that processes this data. Your code must work for *any* valid lists of stations, travel classes, and prices following this structure. (You can refer to the **Example Code** and **Sample Output** sections at the end of this question for reference.)

```
stations: list[str] = [...]  # list of station names (strings)
classes: list[str] = [...]   # list of travel class names (strings),
                             # ordered from least to most luxurious
prices: list[list[int]] = [  # 2D list of positive integers (HKD)
    [...], # prices[i][j] is the fare to stations[i] in classes[j]
    ...
]
```

In this question, you may assume that:

- `stations` and `classes` are non-empty lists of strings.
- To ensure comparisons and ratios are mathematically meaningful, you may assume that `len(classes) >= 2` (i.e., every train has at least two travel classes).
- `prices` is a 2D list of positive integers, where `len(prices) == len(stations)` and every row has exactly `len(classes)` elements.
- The entries of `classes` are always ordered from the least luxurious travel class to the most luxurious travel class (e.g., `"Second"`, `"First"`, `"Premium"`, `"Business"`).

(a) [2 points] The customer service team needs a tool to quickly quote ticket prices to passengers. Implement the function:

```
def part_a(stations: list[str], classes: list[str], prices: list[list[int]],
           input_station: str, input_class: str) -> int:
```

that returns the ticket price of `input_station` in `input_class` from the price table. You may assume that `input_station` and `input_class` are always valid entries in `stations` and `classes`.

(b) The railway network is expanding, and the system administrator needs to update the system with new routes. Implement the function:

```
def part_b(stations: list[str], prices: list[list[int]],
           new_station: str, new_price: str) -> None:
```

that handles the following:

(i) [1 point] Append `new_station` to the `stations` list.

(ii) [3 points] Parse `new_price` (a comma-separated string of integers) into a list of integers and append it as a new row to the `prices` list. You may assume that `new_price` is valid; i.e., it contains exactly one comma-separated integer for each travel class (e.g., `"78,125,140,234"`).

(c) [6 points] The quality assurance (QA) team wants to automatically detect data entry errors in the system. Implement the function:

```
def part_c(stations: list[str], prices: list[list[int]]) -> list[str]:
```

that identifies stations with valid pricing. A station's pricing is considered *valid* only if its fares are strictly increasing as the travel class becomes more luxurious (e.g., $[120, 180, 250]$ is valid, but $[120, 180, 180]$ and $[120, 180, 150]$ are not).

The function should return a list containing the names of all stations that pass this QA check. You may assume all price entries are positive integers. You are **not** allowed to use the `sort()` or `sorted()` functions.

(d) Implement the function:

```
def part_d(stations: list[str], prices: list[list[int]]) \
           -> tuple[list[float], list[str], float]:
```

that computes and returns a tuple `(avg_fare, promo_stations, min_multiplier)`. The function body should contain **exactly FOUR statements**: three Python assignment statements to compute the three values, followed by one return statement. Each assignment must compute its value in a single line of code without using semicolons (;). You may use built-in functions like `sum()` and `min()` as needed:

(i) [3 points] `avg_fare (list[float])`: a list where the i-th element is the average ticket price across all travel classes for `stations[i]`. The railway management team needs this quick summary of general pricing for each destination.

(ii) [3 points] `promo_stations (list[str])`: a list of station names where the average ticket price is strictly less than `HKD 200`, for a promotional campaign targeting cheaper destinations.

(iii) [4 points] `min_multiplier (float)`: the **smallest** upgrade multiplier among all stations. The *upgrade multiplier* for a station is defined as the price of its most luxurious class divided by the price of its least luxurious class. A smaller multiplier indicates a better relative deal for an upgrade. Your code should work regardless of the total number of travel classes.

Example Code. Below is an example code showing how to structure and call your functions. The concrete values shown are for illustration only. Your code must work correctly for *any* lists that satisfy this structure, not just for this particular example.

```python
# Define the functions above here (part_a, part_b, part_c, part_d)
# 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: ")

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

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

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

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

# Call part (d)
avg_fare, promo_stations, min_multiplier = part_d(stations, prices)
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}")
```

Sample Output. Upon completing all parts, your code should produce the following output. In the transcript below, any text after > indicates user input.

```
Enter the station name: >Futian
Enter the class name: >First
(a) fare: 125
Enter new station name: >Humen
Enter prices for Humen (comma-separated): >204,308,428,204
(b)(i) stations: ['Futian', 'Humen']
(b)(ii) prices: [[78, 125, 140, 234], [204, 308, 428, 204]]
(c) valid_stations: ['Futian']
(d)(i) avg_fare: [144.25, 286.0]
(d)(ii) promo_stations: ['Futian']
(d)(iii) min_multiplier: 1.00
```

Problem 8 [16 points] Strings and Lists Utilities

Strings and lists are often utilized in Python programs. Your TA Sunny has asked you to help him create some string and list utilities. **For each part, you can assume functions of previous parts are already correctly implemented and use them in your answers**. Your implementation should always work with input of varying lengths, sizes and dimensions unless mentioned otherwise.

Note: "Primary diagonal" in the question below refers to the diagonal from the top-left to the bottom-right.

(a) (i) [3 points] Implement the function `join`, where it concatenates a one-dimensional list into a string with a separator. If the given list is empty, return an empty string. Examples

- `join([1, 2, 3], "C")` returns `"1C2C3"`
- `join(['h', 'a'], "ah")` returns `"haha"`

You can assume the given list (`target`) will always be one-dimensional, and that all elements within the list can be converted to a string explicitly using the built-in `str` function. **You are NOT allowed to call ANY functions in your code, except the built-in function `str()` in this part**.

(ii) [3 points] Implement the function `get_diagonal`, where it returns the primary diagonal of a **square** list. If the given list is not square, return an empty list. Examples:

- `get_diagonal([[1, 3], [2, 4]])` returns `[1, 4]`
- `get_diagonal([[1, 2, 3], [2, 4, 6, 8]])` returns `[]`

You can assume the given list (`target`) will never be empty nor one-dimensional, i.e., `len(target)` will always be positive and all elements of `target` are lists.

(iii) [2 points] Implement the function `diag_to_str` **using only ONE statement**, where it concatenates the primary diagonal of a **square** list into a string with commas (`,`) as the separator. If the given list is not square, return an empty string. Examples:

- `diag_to_str([[1, 3], [2, 4]])` returns `"1,4"`
- `diag_to_str([["a", 4, 6.3], [1, 2, 3], ['d', "def", 'F']])` returns `"a,2,F"`

Similar to the previous parts, you can assume the given list (`target`) will never be empty nor one-dimensional, i.e., `len(target)` will always be positive and all elements of `target` are lists. You can also assume that all elements within the sub-lists can be converted to a string explicitly using the built-in `str` function. **However, you are NOT allowed to use semi-colons, loops or comprehensions.**

Now, Sunny gives you a challenge: **For the upcoming functions, you can only use ONE statement, and no semi-colons should be used.** Certain functions may also come with additional criteria. Just like before, your implementation should always work with input of varying lengths, sizes and dimensions unless mentioned otherwise.

(b) (i) [4 points] Implement the function `mask_diag` that masks the lower left portion of a square list by converting them to a default value `val`. Specifically, it replaces all elements "below" the primary diagonal to `val`. This function should return a **new** 2-dimensional list with the same dimensions as `target`, except that all elements below the main diagonal are replaced by `val`. Below are some examples given the function call `mask_diag(target, val)`:

target	val	Result

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

`[["a","b"],` `["c","d"]]`	`"haha"`	`[["a","b"],` `["haha","d"]]`
`[[4,5],` `[5,4]]`	`5`	`[[4,5],` `[5,4]]`

You can assume `target` is two-dimensional, i.e., each element of the sub-lists of `target` will not be a collection. For simplicity, you can also assume `target` is a square list.

(ii) [4 points] Implement the function `skip_list`. Given a positive integer `factor`, return a **copy** of `target` where every `factor` element is taken and appended to the back. You must implement this function using **at most ONE** comprehension. Examples:

- `skip_list([1, 2, 3, 4, 5, 6], 3)` returns `[1, 2, 4, 5, 3, 6]`
- `skip_list([1.0, 2.3, 1, 2.3, 1.0], 2)` returns `[1.0, 1, 1.0, 2.3, 2.3]`
- `skip_list(["a", "b"], 1000)` returns `["a", "b"]`

You can assume the given list (`target`) will always be one-dimensional and `factor` is a positive integer.

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

Appendix:

Operator Precedence from High (Top) to Low (Down)

Operator	Description	Associativity
`**`	Exponentiation	Right-to-left
`+, -`	Unary plus and minus	Right-to-left
`*, /, //, %`	Multiplication, division, integer division, and modulus	Left-to-right
`+, -`	Binary addition and subtraction	Left-to-right
`in, not in, is, is not, <, <=, >, >=, !=, ==`	Relational operators (including membership tests)	Left-to-right
`not`	Logical negation	Left-to-right
`and`	Logical conjunction	Left-to-right
`or`	Logical disjunction	Left-to-right
`=, +=, -=, *=, /=, //=, %=, **=`	Assignment and augmented assignment operators	

typing.Any:

Within Python's type hinting system, `typing.Any` denotes a type that is unconstrained and can represent any possible value when applied to a variable, parameter, or return annotation.

Python Function Documentation

- Useful functions
  - `enumerate(iterable, start=0)`
    Add a counter to an `iterable` (such as list) and returns an enumerate object which can be used in loops to access both the index and the value of each item.
  - `len(object)`
    Return the length (the number if items) of an object. The argument may be a sequence (such as a string, bytes, tuple, list of range) or a collection (such as a dictionary, set, or frozen set).
  - `min(iterable)`
    Return the smallest item in an iterable.
  - `range(start, stop[, step])`
    Generate a sequence from `start` (default is 0) up to (but not including) `stop`, incrementing by `step` (default is 1).
  - `sum(iterable, start=0)`
    Sum `start` and the items of an `iterable` from left to right and returns the total. The `iterable`'s items are normally numbers, and the start value is not allowed to be string.

- * operator
  When * operator is used directly on a list and an integer `n`, it performs repetition, creating a new list that repeats the original elements `n` times.

- `list.append(value)`
  Add an item to the end of the list.

- `list.count(value)`
  Return the number of times `value` appears in the list.

- `list.index(value[, start[, stop]])`
  Return zero-based index of the first occurrence of `value` in the list. Raises a `ValueError` if there is no such item. The optional arguments `start` and `end` are interpreted as in the slice notation and are used to limit the search to a particular subsequence of the list. The returned index is computed relative to the beginning of the full sequence rather than the `start` argument.

- `list.pop(index)`
  Remove and return the item at the given `index` from the list.

- `random.shuffle(x)`
  Shuffle the sequence `x` in place. For example:

```
import random
my_list = [1, 2, 3, 4, 5]
random.shuffle(my_list)
print(my_list) # Output will be a randomized version of the list,
               # e.g., [1, 3, 5, 4, 2]
```

- `str.split(sep=None, maxsplit=-1)`
  Return a list of the words in the string, using `sep` as the delimiter string. If `maxsplit` is given, at most `maxsplit` splits are done (thus, the list will have at most `maxsplit+1` elements). If `maxsplit` is not specified or `-1`, then there is no limit on the number of splits (all possible splits are made).

  If `sep` is given, consecutive delimiters are not grouped together and are deemed to delimit empty strings (for example, `'1,,2'.split(',')` returns `['1', '', '2']`). The `sep` argument may consist of multiple characters (for example `'1<>2<>3'.split('<>')` returns `['1', '2', '3']`). Splitting an empty string with a specified operator returns `['']`.

  If `sep` is not specified or is `None`, a different splitting algorithm is applied: runs of consecutive whitespace are regarded as a single separator, and the result will contain no empty strings at the start or end if the string has leading or trailing whitespace. Consequently, splitting an empty string or a string consisting of just whitespace with a `None` separator returns `[]`.

/* Rough work */

/* Rough work */

/* Rough work */

/* Rough work */