COMP 1023 Midterm Exam - Fall 2025 - HKUST
Question Book

Date: October 25, 2025 (Saturday)
Time Allowed: 2 hours, 1:00–3:00 pm
Instructions:

1. This is a closed-book, closed-notes examination.
2. There are **6** questions on **20** pages (including this cover page, appendix and 4 blank pages at the end).
3. Write your answers in the space provided in the answer book in black/blue ink. *NO pencils please, otherwise you are not allowed to appeal for any grading disagreements.*
4. All programming codes in your answers must be written in the Python version as taught in the class.
5. For programming questions, 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	

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) `int` and `float` objects are mutable, meaning their contents can be updated.

(b) The output of the following program is `1023`.

```
def main() -> None:
    a: int = 10
    b: int = 23
    a, b = b, a
    print(a, b, sep="")

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

(c) This code reports an error.

```
def main() -> None:
    x: int = 0
    if x and 10 / x > 1:
        print("Yes")
    else:
        print("No")

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

(d) The output of the following program is `haha`.

```
def main() -> None:
    my_var: None = None
    if my_var == None:
        print("h", sep='a', end='a')
    if my_var is None:
        print("h", sep='a', end='a')
    print()

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

(e) The output of the following program is `Success`.

```
def main() -> None:
    counter: int = 0
    while counter < 3:
        counter += 1
        if counter == 3:
            break
    else:
        print("Success")

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

(f) The output of the following program is `13`.

```
def main() -> None:
    for x in "COMP1023":
        if x.isdigit() and int(x) % 2:
            print(x, end='')
    print()

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

(g) The output of the following program is `F`.

```
def main() -> None:
    print('T' if type(range(5)[0:2]) == type([]) else 'F')

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

(h) If `x` is a list with 5 elements, the slice operation `x[10:]` will raise an error and cannot be executed.

(i) If a Python function does not explicitly return a value, it returns `None` by default.

(j) The output of the following program is `[[99, 2, 3], [4, 5, 6]]`.

```
import copy

def main():
    original_list = [[1, 2, 3], [4, 5, 6]]
    shallow_copied_list = copy.copy(original_list)
    shallow_copied_list[0][0] = 99
    print(original_list)

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

Problem 2 [10 points] Branching and Looping Statements

Analyze the function provided below, which incorporates the use of `for` loop, `continue`, `break`, and multiple `if` statements.

```
def branch_and_loop(nums: list[int | None]) -> int:
    total: int = 1
    for value in nums:
        if value is None:
            total += 2
            continue

        if value % 3 == 0:
            total *= value
        elif value % 2 == 0:
            total += value
            break
        else:
            total -= value

    return total
```

(a) [7 points] Given the list `data = [1, None, 3, 1, -7, -4, 5]`, when we call the function `branch_and_loop(data)`, please estimate the value of `total` after processing **each element** in the list by completing the given table in the answer book. "After processing each element" means the value of `total` is updated based on the current element in the list, and the new value of `total` is recorded before moving on to the next element. Continue this until the loop terminates. If an element is not processed, please put "/" in the table cell.

(b) [3 points] Consider the list `data2 = [None, 6, 9]`. When we execute the function `branch_and_loop(data2)`, please write down the return value of the function. Also, explain the reason for the termination of the loop within the function: will it stop because of a `break` statement, will it run out of items naturally, or is there another reason?

Problem 3 [18 points] Functions and Lists

(a) [11 points] A prime number is a natural number greater than 1 that has no positive divisors other than 1 and itself. Please solve the following problems related to determining prime numbers.

(i) [1 point] Implement a Python function `is_divisible(n, k)` that checks whether `n` is divisible by `k`. Both `n` and `k` are positive integers. You can assume `n` is always larger than `k`.

Examples:

- `is_divisible(10, 5)` should return `True`
- `is_divisible(10, 3)` should return `False`

(ii) [4 points] Implement a Python function `is_prime(n)` that checks whether `n` is a prime number using a **for-loop**. You should break the loop **as early as possible** to optimize efficiency. You can assume that `n` is a natural number greater than 1. You **must reuse** the `is_divisible` function from part (a)(i) to simplify your code.

Examples:

- `is_prime(7)` should return `True`
- `is_prime(9)` should return `False`

(iii) [6 points] Implement a Python function `get_primes(n)` using a **for-loop** that returns two outputs: the number of prime numbers in the range `[2, n]` (i.e., greater than or equal to 2 and less than or equal to `n`), and a list of those prime numbers. You should break the loop **as early as possible** to optimize efficiency. You can assume that `n` is a natural number greater than 2. You **must reuse** the `is_prime` function from part (a)(ii) to simplify your code.

Example:

- `get_primes(10)` should return `(4, [2, 3, 5, 7])`

(b) [7 points] You are given a **square** 2D list of size $n \times n$ ($n \geq 1$), which is represented as a list of lists, with each inner list representing a row. For example, the 3×3 square 2D list `[[1, 2, 3], [4, 5, 6], [7, 8, 9]]` can be visualized as follows:

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

Implement a function `rotate_clockwise(lst)` that takes a square 2D list as input and returns a new 2D list representing the original list rotated 90 degrees clockwise. You must use a new 2D list to store the rotated output.

Example:

- Input: `[[1, 2, 3], [4, 5, 6], [7, 8, 9]]`
- Output: `[[7, 4, 1], [8, 5, 2], [9, 6, 3]]`

Problem 4 [20 points] Mini Store System

In this question, you will create several Python functions to simulate a simple in-game store system. A player starts the game with some money (referred to as balance) and a bag that already contains one item. Your goal is to implement functions that help the player check their balance, view their bag, and purchase items from the store.

The main program has already been written for you as follows:

```
def main() -> None:
    balance: int = 500
    bag: list[str] = ["book"]
    print("Welcome to the store!")
    # Print balance
    print_balance(balance)
    # Print bag
    print_bag(bag)

    # Buy a sword
    print("You want to buy a sword.")
    balance, item, amount = purchase_item(balance=balance, item="sword", price=100)
    bag = put_in_bag(bag, item, amount)
    print_bag(bag)

    # Buy 2 health potions
    print("You want to buy 2 health potions.")
    balance, item, amount = purchase_item(balance, "health_potion", 20, amount=2)
    bag = put_in_bag(bag, item, amount)
    print_bag(bag)

    # Buy a golden apple but not enough money
    print("You want to buy a golden apple.")
    balance, item, amount = purchase_item(balance, "golden_apple", 400)
    bag = put_in_bag(bag, item, amount)
    print_bag(bag)

    print("Thank you for your purchase!")
    print_balance(balance)
    print_bag(bag)

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

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

```
Welcome to the store!
Your balance is $500
Your bag contains: book
You want to buy a sword.
Your bag contains: book sword
You want to buy 2 health potions.
Your bag contains: book sword health_potion health_potion
You want to buy a golden apple.
Your bag contains: book sword health_potion health_potion
Thank you for your purchase!
Your balance is $360
Your bag contains: book sword health_potion health_potion
```

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.**

- `print_balance`
- `print_bag`
- `put_in_bag`
- `purchase_item`

(a) [2 points] Implement the function `print_balance` that prints the current balance.

(b) [6 points] Implement the function `print_bag` that prints all items in the player's bag on one line.

(c) [4 points] Implement the function `put_in_bag` that adds a given item to the player's bag multiple times and returns the updated bag. The number of times to add the item is specified by the parameter `amount`.

(d) [8 points] Implement the function `purchase_item` that attempts to buy an item from the store.

- If the player has enough balance, deduct the total cost (`price * amount`) and return the new balance, the item name, and the quantity purchased.
- If the player does not have enough balance, return the balance unchanged, the item name and an amount of 0.

Problem 5 [26 points] Number Guessing Game

Implement a Python number guessing game where the computer generates a random integer between 1 and 100 (inclusive), and the player has up to 10 attempts to guess it.

The program uses a list to manage the game state with the following structure:
`[secret_number, attempts, max_attempts, low_bound, high_bound, game_over]`
where

- `secret_number`: The random integer the player is trying to guess.
- `attempts`: How many valid guesses the player has already made.
- `max_attempts`: The maximum number of guesses allowed (up to 10).
- `low_bound`: The current lowest valid guess; updates to one above any "too low" guess.
- `high_bound`: The current highest valid guess; updates to one below any "too high" guess.
- `game_over`: Whether the game has ended (`True` if won or attempts exhausted, else `False`).

The game intelligently narrows the possible range after each guess. If a guess number is too low, the lower bound increases to one above the guess, if too high, the upper bound decreases to one below the guess number.

Input validation is performed using string methods and conditional checks. For non-integer input, display "Invalid input! Please enter a whole number" and re-prompt without counting the attempt. For integers outside the current valid range (which dynamically updates), display "Please enter a number between {low_bound} and {high_bound}" and re-prompt.

The attempt counter only increments for valid guesses. After each valid guess, provide "Too high!" or "Too low!" feedback. Upon correct guessing, congratulate the player and show the number of attempts used. If all 10 attempts are exhausted, reveal the secret number.

The program is written with five functions, three of which have missing implementations as follows:

```
import random

def initialize_game() -> list[int | bool]:
    """
    Returns:
    list[int | bool]: A list containing the game state with the following
    elements in order:
    secret number, current attempts count, maximum allowed attempts,
    lower bound of range, upper bound of range, and game over status.

    Part (a): Set up and return the initial state for a new game. The function
    should generate a random secret number and initialize all game parameters.
    """
    pass
```

```python
def get_valid_guess(game_state: list[int | bool]) -> int:
    """
    Parameters:
    game_state: list containing current game state

    Returns:
    int: valid user guess within current range

    Part (b): Prompt the user for a guess and validate the input.
    Display the current attempt number and valid range to the user.
    If the input contains non-digit characters, display
    "Invalid input! Please enter a whole number." and re-prompt.
    If the input is an integer outside the current valid range, display
    an error message "Please enter a number between {lower_bound} and {higher_bound}."
    and re-prompt. Only return when a valid integer within the current range
    is provided. The attempt counter should not be incremented in this function.
    """
    pass

def process_guess(guess: int, game_state: list[int | bool]) -> list[int | bool]:
    """
    Parameters:
    guess: int - the user's validated guess
    game_state: list containing current game state

    Returns:
    list[int | bool]: The updated game state, reflecting the incremented attempt
    counter and any adjustments based on the user's guess.

    Part (c): Process the user's guess and update the game state accordingly.
    Increment the attempt counter in the game state.
    If the guess matches the secret number, display a success message
    "Congratulations! You guessed the number {secret number} in {attempt counter}
    attempts!" and mark the game as over. If the guess is too high or too low,
    provide appropriate feedback "Too high!" or "Too low!" to the user.
    Update the range bounds based on the guess to narrow down the possibilities.
    Finally, return the updated game state.
    """
    pass
```

```python
# The following function has been completed for you.
def play_single_round() -> None:
    game_state: list[int | bool] = initialize_game()

    print("Guess the number (1-100)! You have 10 attempts.\n")

    while game_state[1] < game_state[2] and not game_state[5]:
        guess: int = get_valid_guess(game_state)
        game_state = process_guess(guess, game_state)

    if not game_state[5]:
        print(f"Game over! The secret number was {game_state[0]}.")

# The following function has been completed for you.
def main() -> None:
    print("=== Number Guessing Game ===\n")

    while True:
        play_single_round()

        play_again: str = input("\nWould you like to play again? (y/n): ")
        if play_again not in ['y', 'yes']:
            print("Thanks for playing! Goodbye!")
            break
        print("\n" + "="*30 + "\n")

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

The output of the program after completing all the functions should look like this. The underlined text shows what the user inputs.

```
=== Number Guessing Game ===

Guess the number (1-100)! You have 10 attempts.

Attempt 1 (Range: 1-100): abc
Invalid input! Please enter a whole number.
Attempt 1 (Range: 1-100): 4.3
Invalid input! Please enter a whole number.
Attempt 1 (Range: 1-100): 50
Too low!
```

```
Attempt 2 (Range: 51-100): 75
Too high!

Attempt 3 (Range: 51-74): 30
Please enter a number between 51 and 74.
Attempt 3 (Range: 51-74): 60
Too low!

Attempt 4 (Range: 61-74): 64
Congratulations! You guessed the number 64 in 4 attempts!

Would you like to play again? (y/n): n
Thanks for playing! Goodbye!
```

(a) [4 points] Implement the function `initialize_game` following the instructions given in the description.

(b) [11 points] Implement the function `get_valid_guess` following the instructions given in the description.

(c) [11 points] Implement the function `process_guess` following the instructions given in the description.

Problem 6 [16 points] Sales of Car Accessories

After the weird calculation of cost and time taken of car manufacturing (in Lab 2), your manager Sunny has come in with more requests! He has given you a file containing two-dimensional lists of weekly sales of car accessories of the company. Inside this file, you find that the lists are in the following format:

```
week1_sales = [
    [12, 15, 22, 18, 21, 27, 25],
    [24, 25, 23, 16, 28, 16, 13],
    [29, 26, 28, 18, 26, 15, 14],
    [29, 21, 25, 28, 10, 15, 12],
    [22, 13, 24, 26, 15, 25, 23],
    [11, 19, 28, 18, 27, 25, 20],
    [16, 21, 22, 13, 23, 24, 27]
]
```

Upon investigation, you also find another list `product_names`, which tells you which product each row corresponds to:

```
product_names = [
    "Car Paint",
    "Baby Car Seats",
    "EV Charging Cable",
    "Driving Recorder",
    "USB Car Charger",
    "Headrest",
    "'Baby in Car' Sticker"
]
```

You now look forward to Sunny's instructions on what you are supposed to do.

For parts (a) to (c), you must implement the functions using only **ONE** statement. **Please note that partial points will be deducted for using more than one statement, even if you correctly solve the problems. You also must NOT use any lambda functions or semi-colons for these 3 parts.**

(a) [6 points] Sunny asks you to write a function to transpose a given sales list.

Transposing a two-dimensional list means interchanging each of its rows and columns. That means, the first row will become the first column, the second row will become the second column, etc. Implement the function `transpose(sales_list)` that returns a transposed copy of `sales_list` using only **ONE** statement. You can assume all inner lists of `sales_list` have the same length.

As an example, `transpose([[1, 2, 3], [4, 5, 6]])` will result in the two-dimensional list `[[1, 4], [2, 5], [3, 6]]`.

Hint: Use a nested list comprehension to create the transposed list by iterating over the column indices of `sales_list` and collecting the corresponding elements from each row. Here is an example of nested list comprehension:

```
my_list = [[j + 1 + i * 3 for j in range(3)] for i in range(2)]
print(my_list)
# Output:
# [[1, 2, 3],
#  [4, 5, 6]]
```

```
def transpose(sales_list: list[list[int]]) -> list[list[int]]:
```

(b) [5 points] Then, Sunny tells you that each column of the sales lists refers to each day of the week. However, the values are wrongly input so that each week starts from Monday instead of Sunday. Implement the function `shift(sales_list, days)` such that it returns a copy of the `sales_list` with the columns shifted **LEFT** by `days` using only **ONE** statement. **You must not use any lambda functions or semi-colons**. You can assume all inner lists of `sales_list` have the same length.

For examples:

- `shift([[1, 2, 3], [4, 5, 6]], 1)` will result in the two-dimensional list `[[2, 3, 1], [5, 6, 4]]`.
- `shift([[1, 2, 3], [4, 5, 6]], 2)` will result in the two-dimensional list `[[3, 1, 2], [6, 4, 5]]`.

Hint: Use a list comprehension to shift each row in `sales_list` to the left by the specified number of days using slicing to rearrange the elements.

```
def shift(sales_list: list[list[int]], days: int) -> list[list[int]]:
```

(c) [5 points] Sunny gives you a challenge afterwards, to implement a similar function to `shift()`. This function `shift_right(sales_list, days)` does the exact same thing, except the columns are shifted **RIGHT** by `days`. Here are Sunny's requirements: Implement this using only **ONE** statement. In addition to the given constraints, **you also must NOT use any loops or list comprehension**. You can assume all inner lists of `sales_list` have the same length.

For examples:

- `shift_right([[1, 2, 3], [4, 5, 6]], 1)` will result in the two-dimensional list `[[3, 1, 2], [6, 4, 5]]`.
- `shift_right([[1, 2, 3], [4, 5, 6]], 2)` will result in the two-dimensional list `[[2, 3, 1], [5, 6, 4]]`.

Hint: Use the `shift` function in part (b).

```
def shift_right(sales_list: list[list[int]], days: int) -> list[list[int]]:
```

---------- 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
`<, <=, >, >=`	Relational operators	Left-to-right
`==, !=`	Equality operators	Left-to-right
`is, is not`	Identity operators	Left-to-right
`in, not in`	Membership operators	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	Right-to-left*

- Useful functions
  - `len(c)`
    Return the number of items in a container `c`.
  - `type(obj)`
    Return the type of an object `obj`.
- Copy functions
  - `import copy`
    `copy.copy(obj)`
    Return a shallow copy of the object `obj`.
- Random functions
  - `random.randint(a, b)`
    Return a random integer N such that `a` $\leq N \leq$ `b`.
- String methods
  - `str.isdigit()`
    Return `True` if all characters in the string are digits, otherwise return `False`.
- List methods
  - `list.append(x)`
    Add an item `x` to the end of the list. Equivalent to `a[len(a):] = [x]`.
  - `list.extend(iterable)`
    Extend the list by appending elements from the `iterable`.

/* Rough work */

/* Rough work */

/* Rough work */

/* Rough work */