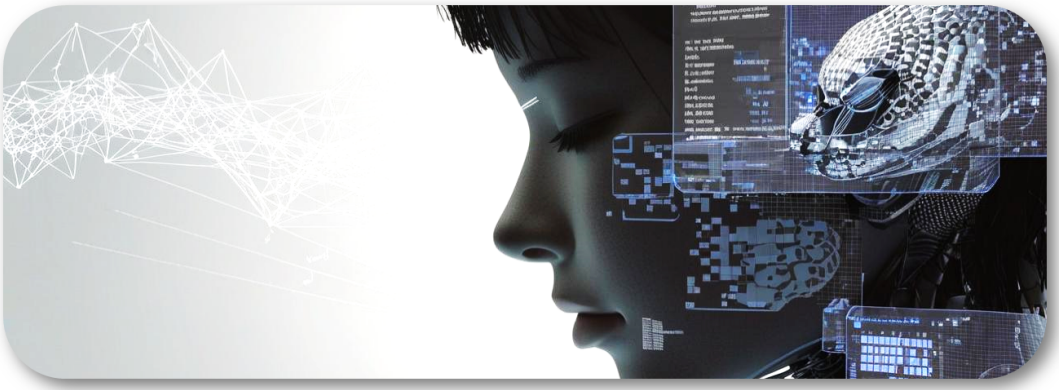


COMP 1023 Introduction to Python Programming

Tutorial 8: Midterm Review & Modules, Packages and Libraries

COMP 1023 Teaching Team

Department of Computer Science & Engineering
The Hong Kong University of Science and Technology, Hong Kong SAR, China

Part I

Solution & Explanation to Selected Midterm Questions

Midterm - Problem 1(j)

This is the last True-or-False subquestion:

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

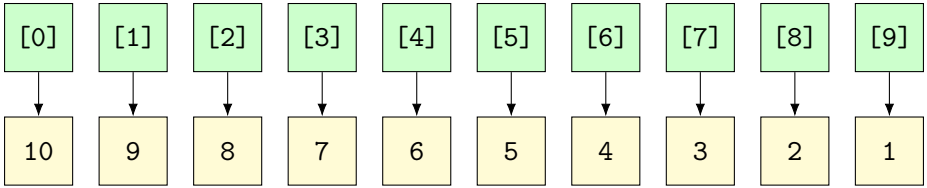
```
import random

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()
```

Solution

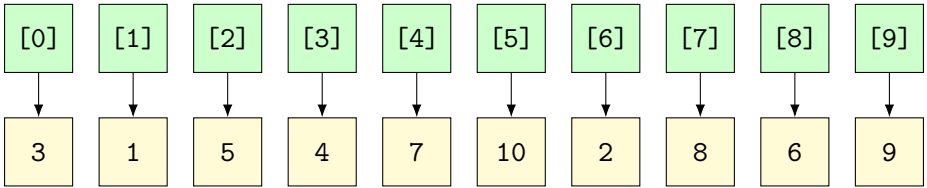
The answer is **F**. But why?

Midterm - Problem 1(j) (Cont.)

Our original `my_list`:

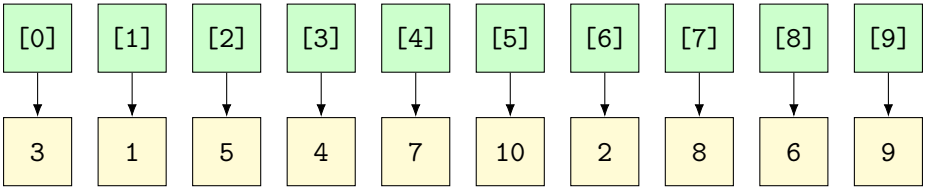


A possible result of `my_list` after `random.shuffle`:



Midterm - Problem 1(j) (Cont.)

Using this result of `my_list`:



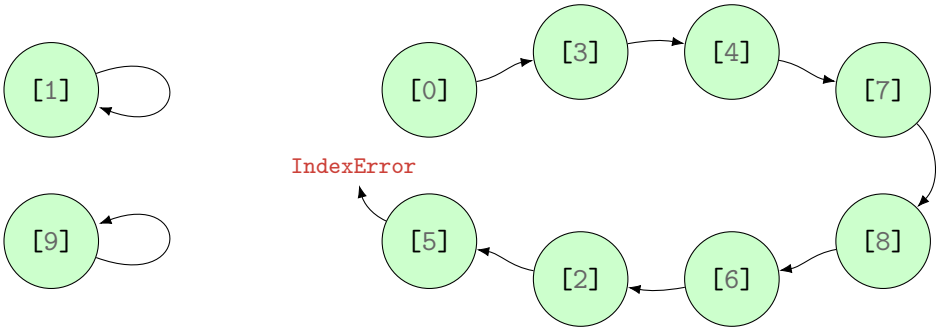
We can acquire the following:

Iteration	x	my_list[x]
0	0	3
1	3	4
2	4	7
3	7	8
4	8	6

Iteration	x	my_list[x]
5	6	2
6	2	5
7	5	10
8	10	IndexError

Midterm - Problem 1(j) (Cont.)

We can redraw the logic like the following:



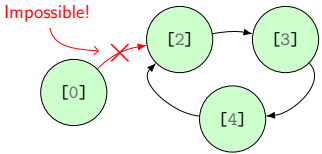
However...

Is it possible for the program to run in an infinite loop? In particular, could we run into a cycle from 0?

Midterm - Problem 1(j) (Cont.)

No! Explanation:

- The number `0` is not in the list.
- No two indices point to the same value, i.e., only one value can lead to a particular index since every number from 1 to 10 appears exactly once in the list.
- There will never be a closed loop.
  - For an infinite loop to occur without returning to 0, `x` would have to enter a cycle (e.g., jump into a repeating circle of indices).
  - Since all values are unique, the entry point of the cycle would need to be reached from two different indices (one from outside the cycle, one from inside), which is impossible! Example:



  - This means `x`, starting from `0`, always hits the "dead-end" that is `10`, raising an `IndexError`.

Midterm - Problem 7(c)

(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.

Midterm - Problem 7(c) (Cont.)

```
def part_c(stations, prices):
    valid_stations = []
    for i in range(len(stations)):
        add = True
        for j in range(len(prices[i]) - 1):
            if prices[i][j] >= prices[i][j+1]:
                add = False
                break
        if add:
            valid_stations.append(stations[i])
    return valid_stations
```

Midterm - Problem 7(c) (Cont.)

```
def part_c(stations, prices):
    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: # Use the for-else clause!
            valid_stations.append(stations[i])
    return valid_stations
```

Other Possible Answers

There are many other possible ways to do this! For example, you could shorten the answer with `enumerate`, or iterate through `stations` in the outer loop!

Midterm - Problem 7(d)

(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.

Midterm - Problem 7(d) (i)

We want to calculate the average fare of each station. First, we need to figure out how to get the average fare of one station.

$$\text{Average fare of a station} = \frac{\text{Sum of prices of a station}}{\text{Number of classes of a station}}$$

Therefore, we can iterate through all the given prices like below:

```
avg_fare = []
for price in prices:
    avg_fare.append(sum(price) / len(price))
```

Since this question require us to complete this in one statement, we can use a list comprehension instead:

```
avg_fare = [sum(row) / len(row) for row in prices]
```

Midterm - Problem 7(d) (ii)

We want to get all stations where the average fare is less than 200. We can reuse the `avg_fare` list we created previously:

```
promo_stations = []
for i in range(len(stations)):
    if avg_fare[i] < 200:
        promo_stations.append(stations[i])
```

Since this question require us to complete this in one statement, we can, once again, use a list comprehension instead:

```
promo_stations = [stations[i] for i in range(len(stations)) if avg_fare[i] < 200]
```

Midterm - Problem 7(d) (iii)

We want to get the smallest upgrade multiplier for a station.

```
min_multiplier = 0
for price in prices:
    upgrade_multiplier = price[-1] / price[0]
    if upgrade_multiplier < min_multiplier:
        min_multiplier = upgrade_multiplier
```

Since this question require us to complete this in one statement, we can use the `min` function:

```
min_multiplier = min([row[-1] / row[0] for row in prices])
```

Midterm - Problem 8(a)(i)

(i) 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**.

Midterm - Problem 8(a)(i) (Cont.)

This is similar to the built-in `str.join` function. Given the constraint of no function calls (except `str()`), this may be difficult.
The easiest way is to add the separator behind each element, and then removing the last separator!

```
from typing import Any
def join(target: list[Any], sep: str) -> str:
    s = ""
    for i in target:
        s += str(i) + sep
    return s[:-len(sep)] # Remove the last n characters,
                         # where n is the length of the separator
```

Midterm - Problem 8(a)(ii)

(ii) 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.

Midterm - Problem 8(a)(ii) (Cont.)

We first need to check if the list is square. Then, we take every diagonal element. When the index of row and column are the same, then it's a diagonal element.

```
from typing import Any
def get_diagonal(target: list[list[Any]]) -> list[Any]:
    row_num = len(target)
    # Check for square
    for i in range(row_num):
        if len(target[i]) != row_num:
            return []
    # Return the diagonal elements
    return [target[i][i] for i in range(row_num)]
```

Midterm - Problem 8(a)(iii)

(iii) 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.**

Midterm - Problem 8(a)(iii) (Cont.)

The difficult constraints here are the one-statement and no-comprehensions requirements. However, if you look closely, you will realize this is a combination of parts (i) and (ii).

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

Midterm - Problem 8(b)(i)

(i) 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`.

Midterm - Problem 8(b)(i) (Cont.)

If you have done part (a)(ii), you can get a similar logic here. All elements "below" the primary diagonal means their row indices is larger than their column indices.

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

Shortening this as a nested comprehension statement:

```
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))
    ]
```

Midterm - Problem 8(b)(ii)

(ii) 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.

Midterm - Problem 8(b)(ii) (Cont.)

We first get each element that is not a multiple of `factor`. Then, we add those that are a multiple at the back:

```
from typing import Any
def skip_list(target: list[Any], factor: int) -> list[Any]:
    ret = []
    for i in range(len(target)):
        if (i + 1) % factor != 0:
            ret.append(target[i])
    for i in range(len(target)):
        if (i + 1) % factor == 0:
            ret.append(target[i])
    return ret
```

Shortening this with list comprehension and slicing:

```
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]
```

Part II

Modules, Packages and Libraries

Modules Overview

- Module: A .py file containing Python code (functions, classes, variables)
- Purpose: Group related code together for reusability
- Built-in Modules: Python comes with many pre-installed modules
- Examples:

```
import math
print(math.sqrt(16))  # Output: 4.0
print(math.pi)        # Output: 3.141592653589793

import random
print(random.randint(1, 10))  # Output: random integer between 1-10
```

Creating Custom Modules (Demo)

- Create your own .py file with functions and variables
- Example: Create `math_operations.py`.

```python
# math_operations.py
def add(a, b):
    return a + b

def subtract(a, b):
    return a - b

def multiply(a, b):
    return a * b

def divide(a, b):
    if b == 0:
        print("Division by zero is not allowed. Returning 0.")
        return 0
    return a / b
```

Importing Modules

- `import module`: Import the entire module, access its names via `module.<name>`
- `import module as m`: Import the entire module, binding the name `m` to `module`
- `from module import item1, item2`: Import specific items directly
- `from module import item1 as name1`: Import specific items with name binding
- `from module import *`: Import all public names of the module

```python
# Method 1: Import entire module
import math_operations
result = math_operations.add(5, 3)

# Method 2: Binding a name to the module
import math_operations as math_op
result1 = math_op.add(5, 3)      # 8
result2 = math_op.multiply(5, 3) # 15

# Method 3: Import specific functions
from math_operations import add, subtract
result1 = add(5, 3)      # 8
result2 = subtract(5, 3) # 2
```

```python
# Method 4: Specific functions + Name binding
from math_operations import subtract as sub,\
    multiply as times
result1 = sub(5, 3)      # 2
result2 = times(5, 5)    # 25

# Method 5: Import all
from math_operations import *
result = multiply(4, 5)  # 20
```

iPRS Question 1

Which of the following import methods is generally discouraged?

A. `import math_operations`

B. `from math_operations import add, subtract`

C. `from math_operations import *`

iPRS Question 1 - Answer

Which of the following import method is generally discouraged?

Explanation

Answer: C. `from math_operations import *`

Shortcomings of `from module import *`:

- Namespace pollution: Hard to track which names are defined and where they come from
- Unintentional Name Shadowing: Imported names may overwrite existing names
- Reduced readability: Which functions or classes from the imported module are being used is unclear

Therefore, in general, the practice of importing * from a module or package is frowned upon

__name__ variable

- Set to "__main__" when file is run as script
- Set to module name when imported
- Example:

```python
# File: script_example.py

print(f"The value of __name__ is: {__name__}")

if __name__ == "__main__":
    # This line only runs when the file is executed directly
    print("This is a script running!")
else:
    print("This file was imported as a module")
```

Importing a Module Multiple Times

- If you import the same module multiple times in the same file, then it will only be run once!

```python
# File: utils.py
def func():
    print("func() called!")

print("utils imported!")
```

```python
# File: main.py
import utils
utils.func()
import utils
utils.func()
```

Output:

```
utils imported!
func() called!
func() called!
```

iPRS Question 2

```python
# File: cost.py
def main():
    print("The total cost will be: $573.52")

main()
```

```python
# File: grader.py
import cost
cost.main()
from cost import main
main()
```

How many times will `The total cost will be: $573.52` be printed when running grader.py?

A. 2

B. 3

C. 4

D. No output due to runtime error

iPRS Question 2

```python
# File: cost.py
def main():
    print("The total cost will be: $573.52")

main()
```

```python
# File: grader.py
import cost
cost.main()
from cost import main
main()
```

How many times will `The total cost will be: $573.52` be printed when running grader.py?

Explanation

Answer: B. 3

The string is printed once when `cost` is imported, then twice more when it is called via `cost.main()` and `main()`. The second import does not rerun the module again.

Packages

- Packages, like modules, are used to organize code in a better way.
- A package is a directory or folder that contains multiple Python modules.
- It is used to group multiple related Python modules together.
- `__init__.py`: A special file that is automatically executed when a package is imported.

Packages (Demo)

```
"""Folder structure:
./
+-- main.py
+-- my_package/
    +-- __init__.py
    +--  module1.py
    +--  module2.py
"""
# File: __init__.py (add the line below to support `from my_package import *`)
__all__ = ["module1", "module2"] # List of all modules
```

```
# File: main.py
from my_package import *  # Import everything specified in __all__
module1.some_function_1()
module2.some_function_2()

from my_package import module1  # Or import specific modules
module1.some_function_1()

from my_package.module1 import some_function_1  # Or import specific functions
some_function_1()
```

Libraries

- A library is a collection of related modules and packages that provide a set of functionalities.
- Libraries are often used to solve specific problems or perform common tasks.
- The term "library" is sometimes used interchangeably with "package", but it generally refers to a larger collection of code.
- Libraries are designed to be reusable across multiple applications.
- Examples of libraries include: `numpy`, `pandas`, `matplotlib` (in-syllabus); `seaborn`, `scipy`, `scikit-learn` (out-of-syllabus).
- The usage is very similar to packages

```
import numpy as np
a = np.array([1, 2, 3, 4])
print(a.mean())  # Output: 2.5
```

Module/Package/Library Exploration Tools

- `dir()`: Lists all attributes and methods in a module
- `__file__`: Path to the module file
- `__doc__`: Module's documentation string
- `help()`: Shows documentation for functions and modules
- Examples:

```
import math
# List all available functions and constants
print(dir(math)) # Output: ['__doc__', 'acos', 'asin', 'atan', 'ceil', 'cos', ...]

# Module information
print(math.__file__)  # Output: (path of the module)
print(math.__doc__)   # Output: This module provides access to...

# Get detailed help
help(math.sqrt)  # Starts an interactive session on the console about math.sqrt()
help(math)       # Starts an interactive session on the console about the math module
```

That's all!

Any questions?

Appendix

- These materials are optional
- Feel free to read them if you are interested

Key Built-in Modules: os and sys

- os module: Operating system interface
- sys module: System-specific parameters and functions

```
import os
print(os.getcwd())          # Get current working directory
os.chdir('/path/to/dir')    # Change directory
print(os.listdir('.'))      # List directory contents

import sys
print(sys.argv)             # Command-line arguments
print(sys.version)          # Python version information
sys.exit(0)                 # Exit program with status code
```

Key Built-in Modules: `math` and `statistics`

- `math` module: Mathematical functions and constants
- `statistics` module: Statistical calculations

```
import math
print(math.pi)           # Output: 3.141592653589793
print(math.pow(2, 3))    # Output: 8.0
print(math.sqrt(25))     # Output: 5.0

import statistics
data = [1, 2, 3, 4, 5]
print(statistics.mean(data))   # Output: 3.0
print(statistics.median(data)) # Output: 3
print(statistics.stdev(data))  # Output: 1.5811388300841898
```

Circular Import Issues

- Circular Import: When two or more modules import each other
- Problem: Creates a dependency loop, causing import errors
- Solutions:
  - Refactor code: Move shared code to separate module
  - Import inside functions: Defer imports until function call

```python
# BAD: Circular import example
# File: user_management.py
from user_validation import is_valid_age
def create_user(name: str, age: int) -> dict:
    if is_valid_age(age): return {"name": name, "age": age, "status": "valid"}
    else: return {"name": name, "age": age, "status": "invalid"}
def get_min_age() -> int:
    return 18
```

```python
# File: user_validation.py
from user_management import get_min_age
def is_valid_age(age: int) -> bool:
    return age >= get_min_age()
```

Solving Circular Imports

Solution 1: Refactor to Separate Module

- Breaks the circle: Creates a linear dependency chain
- Import sequence: shared_module → module_A → module_B (no loop)

```python
# File: user_requirements.py
def get_min_age() -> int:
    return 18
```

```python
# File: user_management.py
from user_validation import is_valid_age
def create_user(name: str, age: int) -> dict[str, str]:
    if is_valid_age(age):
        return {"name": name, "age": str(age), "status": "valid"}
    return {"name": name, "age": str(age), "status": "invalid"}
```

```python
# File: user_validation.py
from user_requirements import get_min_age
def is_valid_age(age: int) -> bool:
    return age >= get_min_age()
```

Solving Circular Imports (Cont.)

Solution 2: Import Inside Functions

- Delays import: Import happens when function is called, not at module load
- Module loading: Both modules can load successfully without immediate import
- Runtime resolution: Import occurs after both modules are already loaded

```python
# File: user_management.py
def create_user(name: str, age: int) -> dict[str, str]:
    from user_validation import is_valid_age  # Import inside function
    if is_valid_age(age):
        return {"name": name, "age": str(age), "status": "valid"}
    return {"name": name, "age": str(age), "status": "invalid"}
def get_min_age() -> int:
    return 18
```

```python
# File: user_validation.py
def is_valid_age(age: int) -> bool:
    from user_management import get_min_age  # Import inside function
    return age >= get_min_age()
```