MTH 3300 STRA Final Exam Key

May 22, 2025

Name	

	Points	**Total Points**
Problem 1		10
Problem 2		10
Problem 3		15
Problem 4		15
Problem 5		10
Problem 6		10
Problem 7		10
Problem 8		20
Total		100

Problem 1

Given two strings `ransom_note` and `magazine`, return `True` if `ransom_note` can be constructed using the letters from `magazine` and `False` otherwise.

Example:

```
can_construct_ransom_note("aa", "aba") outputs True
can_construct_ransom_note("aa", "ab") outputs False
```

```python
def can_construct_ransom_note(ransom_note: str, magazine: str) -> bool:
    freqs = {}

    for char in magazine:
        freqs[char] = freqs.get(char, 0) + 1

    for char in ransom_note:
        if char not in freqs or freqs[char] == 0:
            return False
        freqs[char] -= 1

    return True
```

Problem 2

Remember that a factorial is defined as $n! = n \times n - 1 \times ... \times 1$.

Write a function that finds the sum of the digits of a factorial. For example, $4! = 4 \times 3 \times 2 \times 1 = 24$, so `sum_factorial_digits(4)` would produce 6 since $4! = 24$ and $2 + 4$ would give us 6.

```python
def sum_factorial_digits(n: int) -> int:
    factorial = 1
    for i in range(1, n + 1):
        factorial *= i

    _sum = 0
    while factorial > 0:
        _sum += factorial % 10
        factorial //= 10

    return _sum
```

Problem 3

Write a function `generate_pascal(n: int)` that generates the first n rows of Pascal's Triangle. Each number in the triangle is the sum of the two numbers directly above it.

Example of Pascal's Triangle:

```
Row 0:     1
Row 1:    1 1
Row 2:   1 2 1
Row 3:  1 3 3 1
Row 4: 1 4 6 4 1
```

Your function should return a list of lists, where each inner list represents a row. For example, `generate_pascal(3)` should return:

```
[
    [1],
    [1, 1],
    [1, 2, 1]
]
```

Note: This triangle has many interesting mathematical properties:

- Each row starts and ends with 1
- Each number is the sum of the two numbers above it
- Row n contains the coefficients of the expansion of $(x + y)^n$

```python
def generate_pascal(n: int) -> list[list[int]]:
    if n <= 0:
        return []

    triangle = [[1]]
    for i in range(1, n):
        row = [1]
        for j in range(1, i):
            row.append(triangle[i-1][j-1] + triangle[i-1][j])
        row.append(1)
        triangle.append(row)

    return triangle
```

Problem 4

Each question in this section is worth 3 points.

i) Consider the following code:

```
class A:
    def __init__(self):
        self._x = 1
        self.__y = 2

    def get_y(self):
        return self.__y

class B(A):
    def __init__(self):
        super().__init__()
        self._x = 3
        self.__y = 4

b = B()
print(b._x, b.get_y())
```

What is the output?

The output is `3 2`. The reason is that `_x` is not mangled, so it takes the value from the subclass. `__y` is mangled, so it takes the value from the subclass. Note that name mangling is a mechanism Python uses to make attributes that start with double underscores more "private" by automatically modifying their names.

So `b.__y` is actually `b._B__y`, which is 4, but `b.get_y()` is actually `b._A__get_y()`, which is 2.

ii) Consider this code:

```
class Vector:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __add__(self, other):
        return Vector(self.x + other.x, self.y + other.y)

    def __str__(self):
        return f"({self.x}, {self.y})"


v1 = Vector(1, 2)
v2 = Vector(3, 4)
v3 = v1 + v2
print(v3)
```

What is the output?

The output is `(4, 6)`.

We create two vectors `v1` and `v2` with coordinates `(1, 2)` and `(3, 4)`.

We then add them together to get a new vector `v3` with coordinates `(4, 6)`.

We then print `v3`, which is `(4, 6)`.

iii) Given:

```
class A:
    @classmethod
    def method(cls, x):
        return x

class B(A):
    @staticmethod
    def method(x):
        return x + 1

b = B()
print(b.method(1))
print(B.method(1))
```

What is the output?

The output is `2 2`.

The reason is that `method` is a static method in class `B`, so it takes the class as the first argument.

When we call `b.method(1)`, it actually calls `B.method(B, 1)`.

When we call `B.method(1)`, it actually calls `B.method(B, 1)`.

iv) Consider this code:

```
class A:
    def __init__(self):
        self.value = 0

    def __add__(self, other):
        return self.value + other

    def __sub__(self, other):
        return other - self.value

a = A()
print(a + 5)
print(10 - a)
```

What is the output?

The output is `5 TypeError`

The reason is that `__add__` is a magic method that is called when we use the `+` operator.

When we call `a + 5`, it actually calls `a.__add__(5)`.

But when we do `10 - a`, we are not able to subtract an instance of `A` from an integer. This will raise a `TypeError`

v) Given the following code:

```
class Base:
    multiplier = 2

    def __init__(self, value):
        self.value = value

    @classmethod
    def update_multiplier(cls, new_value):
        cls.multiplier = new_value

    def calculate(self):
        return self.value * self.multiplier

class Derived(Base):
    multiplier = 3

    def calculate(self):
        return super().calculate() + 5

base_obj = Base(10)
derived_obj = Derived(10)

Base.update_multiplier(4)
print(f"Result 1: {base_obj.calculate()}")
print(f"Result 2: {derived_obj.calculate()}")

Derived.update_multiplier(6)
print(f"Result 3: {base_obj.calculate()}")
print(f"Result 4: {derived_obj.calculate()}")
```

What is the output? The output should be:

```
Result 1: 40
Result 2: 35
Result 3: 40
Result 4: 65
```

For `base_obj`, we initially set the value to be 10. The multiplier is initially 2 for the `Base` class.

For `derived_obj`, we initially set the value to be 10. The multiplier is initially 3 for the `Derived` class.

Before we print `Result 1`, we call `Base.update_multiplier(4)`. This changes the multiplier for the `Base` class to be 4.

This means that `Result 1` is `10 * 4 = 40`.

For `Result 2`, we call `derived_obj.calculate()`. This calls `Derived.calculate(derived_obj)`.

The `Derived` class has a `calculate` method that calls the `calculate` method of the `Base` class and adds 5 to the result.

So `Result 2` is `(10 * 3) + 5 = 35`.

Before we print `Result 3`, we call `Derived.update_multiplier(6)`. This changes the multiplier for the `Derived` class to be 6.

However, this does not affect the `Base` class.

So `Result 3` is `10 * 4 = 40`.

For `Result 4`, we call `derived_obj.calculate()`. This calls `Derived.calculate(derived_obj)`.

The `Derived` class has a `calculate` method that calls the `calculate` method of the `Base` class and adds 5 to the result.

So `Result 4` is `(10 * 6) + 5 = 65`.

Problem 5

The greatest common divisor of two or more integers which are not all zero is the largest positive integer that divides each of the integers. For example, the GCD of 8 and 12 is 4.

(a) (5 points) Write a **recursive** function `gcd(x, y)` that computes the GCD of two integers x and y using the Euclidean algorithm. The function should return the GCD of the two numbers. Note that we can use the following piecewise function to represent the GCD.

$$f(x, y) := \begin{cases} x \text{ if } y = 0 \\ f(y, x \bmod y) \text{ otherwise} \end{cases}$$

```
def gcd(x, y):
    if y == 0:
        return x
    return gcd(y, x % y)
```

(b) (5 points) Implement the same method above, but **iteratively**, i.e. use a loop

```
def gcd(x, y):
    while y != 0:
        x, y = y, x % y
    return x
```

Problem 6

Suppose that you have the following data in a CSV file:

```
name,age,species,health_status
Max,3,Dog,Healthy
Luna,2,Cat,Needs Medication
Charlie,5,Dog,Recovering
Bella,1,Cat,Healthy
Rocky,4,Dog,Needs Medication
Milo,2,Cat,Healthy
Shadow,6,Dog,Recovering
Lucy,3,Cat,Healthy
Cooper,2,Dog,Healthy
Lily,4,Cat,Needs Medication
```

Given the following class definition:

```python
class Animal:
    name: str
    age: int
    species: str
    health_status: str

    def __init__(self, name: str, age: int, species: str, health_status: str):
        self.name = name
        self.age = age
        self.species = species
        self.health_status = health_status
```

Add a `classmethod` to the above class called `from_csv(cls, filepath: str) -> list[Animal]` that reads the CSV data from `filepath` and creates an `Animal` instance for each row

```python
import csv


class Animal:
    name: str
    age: int
    species: str
    health_status: str

    @classmethod
    def from_csv(cls, filepath: str) -> list[Animal]:
        with open(filepath, 'r') as file:
            reader = csv.reader(file)
            next(reader)  # Skip header row
            animals = []
            for row in reader:
                animals.append(cls(row[0], int(row[1]), row[2], row[3]))
            return animals
```

Problem 7

Given a list of `float` numbers `nums`, determine if the sequence of numbers is a geometric progression. By definition, a geometric progression is a sequence of numbers that have the form:

$$a, ar, ar^2, ar^3, ar^4 ...$$

where a is the initial value and r is the common ratio.

For example, $2, 6, 18, 54, ...$ is a geometric progression with a common ratio of 3.

```
def is_geometric_progression(nums: list[int]) -> bool:
    ratio, rem = None, None

    for i in range(len(nums) - 1):
        if ratio is None:
            ratio = nums[i+1] // nums[i]
            rem = nums[i+1] % nums[i]
        elif nums[i+1] // nums[i] != ratio or nums[i+1] % nums[i] != rem:
            return False
    return True
```

Problem 8

You are tasked with implementing a basic inventory system for an RPG game. The system should handle items, stacks of items, and inventory management.

Part 1: Item Class (5 points)

Create a class Item with:

- Attributes: `name` (str), `weight` (int), `value` (int), `rarity` (str)

```
class Item:
    name: str
    weight: int
    value: int
    rarity: str

    def __init__(self, name: str, weight: int, value: int, rarity: str):
        self.name = name
        self.weight = weight
        self.value = value
        self.rarity = rarity
```

Part 2: Stack Class (7 points)

Create a class Stack that represents multiple items of the same type:

- Attributes:
  - `item` (Item instance)
  - `quantity` (int, cannot be negative)
- Properties:
  - `total_weight`
  - `total_value`
- Methods:
  - `add(self, n: int)`: increases quantity by n
  - `remove(self, n: int)`: decreases quantity by n (minimum quantity is 0)

```
class Stack:
    item: Item
    quantity: int

    def __init__(self, item: Item, quantity: int):
        self.item = item
        self.quantity = quantity

    @property
    def total_weight(self):
        return self.item.weight * self.quantity

    @property
    def total_value(self):
        return self.item.value * self.quantity

    def add(self, n: int):
        self.quantity += n

    def remove(self, n: int):
        self.quantity = max(0, self.quantity - n)
```

Part 3: Inventory Class (8 points)

Create a class Inventory that manages collections of Stacks:

- Attributes:
  - `maximum_weight` (int)
  - `_items` (list of Stacks, internal use only)
- Property:
  - `total_value`: returns sum of all stack values
- Methods:
  - `add_item(self, item: Item, quantity: int = 1)`:
    - If item exists, increase its stack quantity
    - If item is new, create new stack
  - `get_items_by_rarity(self, rarity: str) -> list[Stack]`:
    - Returns list of all stacks with matching rarity
  - `transfer_to(other: 'Inventory', item_name: str, quantity: int)`:
    - Moves specified quantity of named item to other inventory

Example usage:

```
>>> sword = Item("Steel Sword", weight=5, value=100, rarity="Common")
>>> inv = Inventory(maximum_weight=50)
>>> inv.add_item(sword, 2)
>>> print(inv.total_value)  # Should print: 200
>>> rare_items = inv.get_items_by_rarity("Rare")  # Should return empty list
```

```
class Inventory:
    maximum_weight: int
    _items: list[Stack]

    def __init__(self, maximum_weight: int):
        self.maximum_weight = maximum_weight
        self._items = []

    @property
    def total_value(self):
        return sum(stack.total_value for stack in self._items)

    def add_item(self, item: Item, quantity: int = 1):
        for stack in self._items:
            if stack.item.name == item.name:
                stack.add(quantity)
                return
        self._items.append(Stack(item, quantity))

    def get_items_by_rarity(self, rarity: str) -> list[Stack]:
        return [stack for stack in self._items if stack.item.rarity == rarity]

    def transfer_to(self, other: 'Inventory', item_name: str, quantity: int):
        for stack in self._items:
            if stack.item.name == item_name:
                if stack.quantity > quantity:
                    stack.remove(quantity)
                    other.add_item(stack.item, quantity)
                    return
                else:
                    other.add_item(stack.item, stack.quantity)
                    self._items.remove(stack)
```