Northeastern University

CS 2100: Program Design and Implementation 1

Practice Final Exam

Instructions

- Please put all of your answers on the answer sheet. Only the answer sheet will be graded.
- Do not begin the exam until instructed to do so.
- You may use both sides of a sheet of paper up to 8.5"x11" for reference, but no other resources, including phones, computers, AI, headphones, and ear pods.
- Students may not leave the classroom during the first 10 minutes of the exam (except in case of emergency).
- Hand your completed answer sheet to an instructor before leaving the room.
- Talk to an instructor if you need to leave the room and reenter.

Properties (Revisited from Quiz 2)

```
class BurritoOrder:
    def __init__(self, customer_name, protein):
        self._customer_name = customer_name
        self._protein = protein
        self._tortilla = "Flour"

    @property
    def customer_name(self):
        return self._customer_name.upper()

    @customer_name.setter
    def customer_name(self, value):
        if len(value) > 0:
            self._customer_name = value
        else:
            raise ValueError("Customer name cannot be empty")

    @property
    def tortilla(self):
        return self._tortilla

    @tortilla.setter
    def tortilla(self, value):
        valid_tortillas = ["Flour", "Wheat", "Spinach", "Corn"]
        if value in valid_tortillas:
            self._tortilla = value


o1 = BurritoOrder("Maria", "Tofu")
print(o1.customer_name)  # Line A

o1.tortilla = "Candy"
print(o1.tortilla)  # Line B
```

1. What is the output of Line A?

   a. `maria`

   b. `Maria`

   c. `MARIA`

   d. `maria.upper()`

2. What is the output of Line B?

   a. `Candy`

   b. `Flour`

   c. `None`

   d. (Empty string)

3. Which of the following statements is true?

   a. The `tortilla` property has both a getter and a setter

   b. The `tortilla` property is read-only

   c. The `tortilla` property can be modified by assigning a new value

   d. The `tortilla` property directly accesses the `_tortilla` attribute

4. What happens if you execute `o1.customer_name = ""`?

   a. The `customer_name` is set to an empty string

   b. The `customer_name` remains unchanged

   c. A `ValueError` is raised

   d. The `customer_name` is set to `None`

Lists (Revisited from Quiz 2)

```
class Ingredient:
    """An ingredient with a name, category, and price"""
    def __init__(self, name: str, category: str, price: float):
        if price < 0:
            raise ValueError("Price cannot be negative")
        self.name = name
        self.category = category
        self.price = price

    def __str__(self) -> str:
        return f"{self.name} ({self.category}): ${self.price:.2f}"

class Burrito:
    """A burrito with ingredients"""
    def __init__(
            self, base_price: float = 7.50,
            ingredients: list[Ingredient] = []
    ) -> None:
        if base_price < 0:
            raise ValueError("Base price cannot be negative")
        self.base_price = base_price
        self.ingredients = ingredients

    def add_ingredient(self, ingredient: Ingredient) -> None:
        """Adds an ingredient to the burrito"""
        self.ingredients.append(ingredient)

def main() -> None:
    rice = Ingredient("Brown Rice", "base", 0.00)
    beans = Ingredient("Black Beans", "protein", 0.00)
    chicken = Ingredient("Grilled Chicken", "protein", 2.50)
    guacamole = Ingredient("Guacamole", "topping", 2.00)
    cheese = Ingredient("Cheese", "topping", 1.00)
    salsa = Ingredient("Pico de Gallo", "topping", 0.50)

    burrito = Burrito(base_price=7.50)
    burrito.add_ingredient(rice)
    burrito.add_ingredient(beans)
    burrito.add_ingredient(chicken)
    burrito.add_ingredient(guacamole)
    burrito.add_ingredient(cheese)
```

5. How can I get a list of the names of all ingredients in `burrito` which are free (cost 0.00)?

a. `[ing.name for ing in burrito.ingredients if ing.price == 0.00]`

b. `[ing for ing in burrito.ingredients if ing.price == 0.00]`

c. `[ing.name if ing.price == 0.00 for ing in burrito.ingredients]`

d. `[ing.name for ing in burrito.ingredients if ing == 0.00]`

6. How can I get the list of ingredients in `burrito`, sorted by price from lowest to highest?

a. `sorted(burrito.ingredients, key=lambda ing: ing.price)`

b. `sorted(burrito.ingredients, key=ing.price)`

c. `sorted(burrito.ingredients, key=lambda ing: price)`

d. `sorted(burrito.ingredients.price, key=lambda ing: ing)`

7. How can we check whether any ingredient in `burrito` costs more than 2.00? (Recall: `any(sequence)` returns `True` if any element in `sequence` is `True`, and `False` if all elements are `False`.)

a. `any(ing.price > 2.00 for ing in burrito.ingredients)`

b. `any(ing for ing in burrito.ingredients if ing.price > 2.00)`

c. `any(ing > 2.00 for ing in burrito.ingredients)`

d. `any(ing.price for ing in burrito.ingredients)`

8. How can I get the total number of `"topping"` ingredients in `burrito`?

a. `len([ing for ing in burrito.ingredients if ing.category == 'topping'])`

b. `len([ing.category for ing in burrito.ingredients if ing == 'topping'])`

c. `len([ing if ing.category == 'topping' for ing in burrito.ingredients])`

d. `count([ing for ing in burrito.ingredients if ing.category == 'topping'])`

Abstract methods (Revisited from Quiz 3)

```
from abc import ABC, abstractmethod

class MenuItem:
    """Represents a menu item with a name and item ID."""

    def __init__(self, name: str, item_id: int):
        self.name = name
        self.item_id = item_id
        self.dietary_tags: set[str] = set()


class CustomizableItem(MenuItem, ABC):
    """Represents a customizable menu item with modification count."""

    def __init__(
            self, name: str, item_id: int,
            max_modifications: int):
        super().__init__(name, item_id)
        self.max_modifications = max_modifications

    @abstractmethod
    def apply_customization(self, toppings: set[MenuItem]) -> str:
        pass


class Burrito(CustomizableItem):
    """Represents a burrito with topping tracking."""

    def __init__(
        self, name: str, item_id: int, max_modifications: int,
        available_toppings: set[MenuItem],
    ):
        super().__init__(name, item_id, max_modifications)
        self.topping_usage = {
            topping: 0 for topping in available_toppings}

    def apply_customization(self, toppings: set[MenuItem]) -> str:
        for topping in toppings:
            if topping in self.topping_usage:
                self.topping_usage[topping] += 1
        return f"Added {len(toppings)} toppings"
```

9. Can we instantiate a `MenuItem`?

a. No, because it has an abstract method.

b. Yes, because it has a constructor. If we hadn't written a constructor, then it would not be possible to instantiate.

c. Yes, because it has no abstract methods.

d. Yes, because it has no concrete methods.

10. What happens if we create a `class Taco(CustomizableItem)`, but we don't implement `apply_customization()`?

a. Python will automatically generate a default implementation.

b. The class will work fine as long as we never call `apply_customization()`.

c. We cannot instantiate `Taco` because it still has an abstract method.

d. The `@abstractmethod` decorator is inherited, making `Taco` concrete anyway.

11. Which of the following statements is true about the `CustomizableItem` class?

a. It cannot be subclassed because it has an abstract method.

b. All of its concrete subclasses must implement `apply_customization()`.

c. It can be instantiated if we remove the `ABC` from its inheritance.

d. It requires all subclasses to also be abstract classes.

12. Can we instantiate a `CustomizableItem`?

a. No, because it has an abstract method.

b. Yes, because it has a constructor. If we hadn't written a constructor, then it would not be possible to instantiate.

c. Yes, because it has no abstract methods.

d. Yes, because it has no concrete methods.

Inheritance (Revisited from Quiz 3)

(Same code snippet as the Abstract methods topic)

13. Is this legal:

```
print(Burrito("Chicken Burrito", 6, 5, set()).dietary_tags)
```

a. No, because `Burrito` does not have a `dietary_tags` attribute.

b. Yes, because `Burrito` inherits `dietary_tags` from `MenuItem`.

c. No, because `CustomizableItem` blocks `Burrito` from inheriting `dietary_tags` from `MenuItem`.

d. Yes, but `dietary_tags` is always `None`.

14. If we modify `Burrito`'s `apply_customization()`, will `CustomizableItem`'s `apply_customization()` also be automatically updated?

a. Yes, because `Burrito` overwrites `apply_customization()`.

b. Yes, because `Burrito`'s `apply_customization()` calls `super().apply_customization()`.

c. No, because `Burrito` is not a subclass of `CustomizableItem`.

d. No, because `CustomizableItem` is not a subclass of `Burrito`.

15. Suppose we add this method to `CustomizableItem`:

```
def get_item_name(self) -> str:
    return self.name
```

Can we call this method on a `Burrito` object?

a. No, because `Burrito` doesn't define `get_item_name()`.

b. Yes, because `Burrito` inherits it from `CustomizableItem`.

c. No, because `name` is defined in `MenuItem`, not `CustomizableItem`.

d. Yes, but only if we override it in `Burrito`.

16. If `Burrito`'s `__init__()` didn't call `super().__init__()`, what would change?

a. Nothing, because `super().__init__()` is optional.

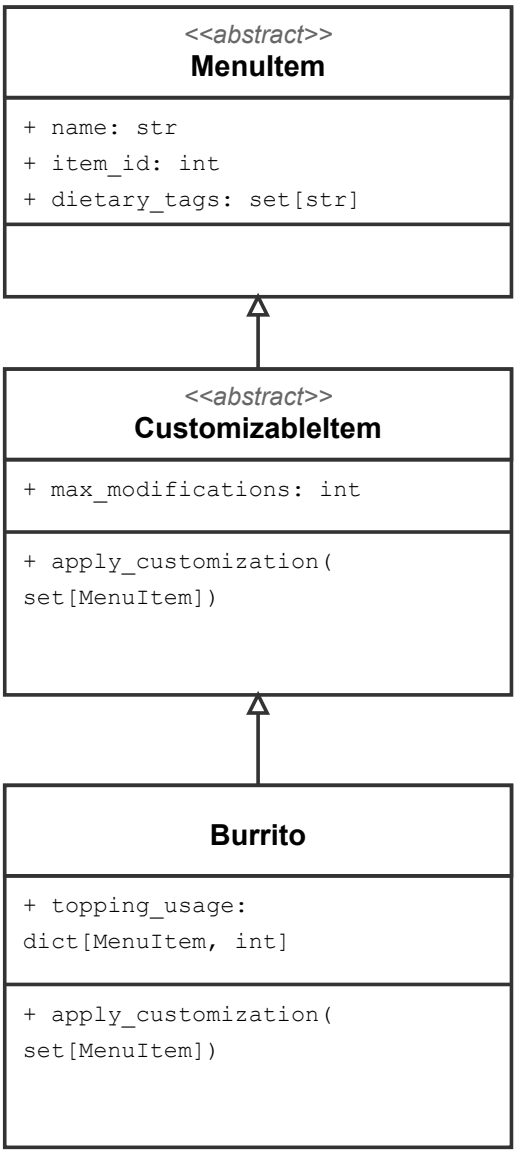
b. The `max_modifications` attribute would not be initialized.

c. Python would automatically call the parent constructors anyway.

d. The `Burrito` class would become abstract.

UML Diagrams (Revisited from Quiz 3)

Consider this UML diagram for an online burrito ordering system, along with the code provided in the Abstract methods topic above:



17. What part of the UML diagram does not match the provided code?

    a. It says that `MenuItem` is abstract, but it shouldn't be.

    b. It says that `CustomizableItem` is abstract, but it shouldn't be.

    c. It says that `Burrito` is abstract, but it shouldn't be.

    d. It says that `Burrito` is not abstract, but it should be.

18. What is wrong with the UML diagram (other than that it doesn't match the provided code)?

    a. `MenuItem`'s item_id is missing a type.

    b. `MenuItem` has no methods.

    c. `MenuItem` has no attributes.

    d. `apply_customization()` appears multiple times.

19. If we wanted to add a `calculate_price()` method to the UML diagram that appears only in `CustomizableItem`, where should it be placed?

a. In the top section of the `CustomizableItem` box.

b. In the middle section of the `CustomizableItem` box.

c. In the bottom section of the `CustomizableItem` box.

d. Outside the `CustomizableItem` box with an arrow pointing to it.

20. If we wanted to show that `Burrito` has a constructor parameter `available_toppings` of type `set[MenuItem]` in the UML diagram, where would this information typically appear?

a. In the attributes section, since constructor parameters become attributes.

b. In the methods section as `__init__(available_toppings: set[MenuItem])`.

c. In a separate "Constructor" section between attributes and methods.

d. UML diagrams don't typically show constructor parameters explicitly unless they become attributes.

Privacy (Revisited from Quiz 3)

21. Which of the following statements is true?

a. Users always know exactly who will receive their personal information when they place an order.

b. Ordering platforms are legally required to delete all user data immediately after each order is completed.

c. Metadata (such as order time, location, and frequency) can reveal personal patterns even without knowing what was ordered.

d. If data is anonymized, it can never be traced back to an individual user.

22. Considering an online burrito ordering and delivery platform, who is the subject of the information?

a. The burrito

b. The restaurant

c. The customer

d. The delivery worker

23. Considering the online burrito ordering platform, who is NOT a likely recipient of the information?

a. The restaurant preparing the order

b. The delivery driver

c. People standing in the burrito restaurant watching delivery workers pick up orders

d. Other customers of the burrito ordering platform

24. The burrito platform wants to use order history to send personalized promotions. Which practice best preserves the users' privacy (by reducing the number of unintended recipients)?

a. Send promotions to all users without asking, since they already have accounts.

b. Sell the order history to third-party marketing companies.

c. Allow users to opt in or opt out of personalized marketing.

d. Share individual order details on social media to show popular items.

Coupling / cohesion / encapsulation (Revisited from Quiz 4)

```
class Order:
    """A burrito order in an online ordering system."""

    def __init__(
        self, customer_name: str, burrito_type: str, spice_level: int, toppings: list[str
    ) -> None:
        self.customer_name = customer_name
        self.burrito_type = burrito_type
        self.spice_level = spice_level
        self.toppings = toppings

    def __eq__(self, value: object) -> bool:
        if not isinstance(value, Order):
            return False
        return self.customer_name == value.customer_name and self.burrito_type == value.b

    def __ne__(self, value: object) -> bool:
        if not isinstance(value, Order):
            return True
        return self.customer_name != value.customer_name or self.burrito_type != value.bu

    def __gt__(self, other: "Order") -> bool:
        if self.burrito_type == other.burrito_type:
            return self.spice_level > other.spice_level
        return self.burrito_type > other.burrito_type

    def __str__(self) -> str:
        return f'{self.customer_name}: spice level {self.spice_level}, {" ".join(self.top

    def send_confirmation_email(self) -> None:
        print(f"Email sent to customer for order: {str(self)}")


class KitchenQueue(Iterable[Order]):
    """Queue of orders filtered by available ingredients."""

    def __init__(self, orders: list[Order], available_ingredients: list[str]) -> None:
        self.orders = orders
        self.available_ingredients = available_ingredients

    def __iter__(self) -> Iterator[Order]:
        return KitchenQueueIterator(sorted(self.orders), self.available_ingredients)
```

```
class KitchenQueueIterator(Iterator[Order]):
    """Iterator for KitchenQueue that yields orders matching available ingredients."""

    def __init__(self, orders: list[Order], available_ingredients: list[str]) -> None:
        self.orders = orders
        self.available_ingredients = available_ingredients
        self.position = 0

    def __next__(self) -> Order:
        while self.position < len(self.orders):
            current_order = self.orders[self.position]
            self.position += 1
            if all(topping in current_order.toppings
                   for topping in self.available_ingredients):
                return current_order
        raise StopIteration
```

25. What would be most appropriate to improve the cohesion of the `Order` class?

    a. Remove the `send_confirmation_email` method, because sending emails is not related to storing order info.

    b. Rename the `send_confirmation_email` method to `notify_customer`.

    c. Shrink the size of the message sent by `send_confirmation_email` to use fewer attributes.

    d. Make `send_confirmation_email` also return the string, in addition to printing it.

26. What would be most appropriate to improve the encapsulation of the `Order` class?

    a. Remove the `self.spice_level` attribute.

    b. Rename `self.spice_level` to have an underscore, and add a method `adjust_spice()` which updates it.

    c. Mention the spice level in the class documentation.

    d. Replace `self.spice_level` with two attributes: `self.public_spice_level` and `self.private_spice_level`.

27. What would be most appropriate to reduce the coupling between `Order` and `KitchenQueueIterator`?

    a. Make it so the `__next__` method uses more of `Order`'s attributes.

    b. Make it so the `__next__` method does not directly access any of `Order`'s attributes.

    c. Make it so the `KitchenQueueIterator` does the sorting in its constructor, instead of the sorting being done in `KitchenQueue`'s `__iter__` method.

    d. Make it so the `__next__` method checks that the orders are sorted, and raises an error if they are not.

28. If we add a method `calculate_total_price()` to the `Order` class, how does this affect cohesion?

a. It decreases cohesion, because pricing is a separate concern from order data.

b. It increases cohesion, because pricing is closely related to order information.

c. It has no effect on cohesion, only on coupling.

d. It violates encapsulation by exposing internal calculations.

Iterator (Revisited from Quiz 4)

(Use the code provided in the "Coupling / cohesion / encapsulation" section)

29. What happens if none of the orders in the `KitchenQueueIterator` match all of the kitchen's available ingredients?

a. A loop over the orders will raise an error (other than `StopIteration`).

b. A loop over the orders will iterate forever.

c. A loop over the orders will `raise StopIteration` on the first iteration.

d. A loop over the orders will return `None` before quitting.

30. What happens if you iterate over the same `KitchenQueue` object twice?

```
kitchen_queue = KitchenQueue([Order('Alice', 'chicken', 2, [])], [])
[order for order in kitchen_queue]  # First iteration
[order for order in kitchen_queue]  # Second iteration
```

a. Both iterators will work correctly and return the same orders

b. The second iterator will be empty because the iterator is exhausted

c. A `RuntimeError` will be raised on the second iteration

d. The second iterator will continue from where the first one ended

31. Why does the `KitchenQueue` class inherit from `Iterable[Order]`?

a. To be able to store a collection of `Order` objects

b. To make it so that `KitchenQueue` can be used in a for loop

c. To indicate to a reader that `KitchenQueue` can be used in a for loop and must implement `__iter__`

d. To automatically generate an `__iter__` method

32. What is the purpose of the while loop in the `__next__` method?

a. To sort the orders before returning them

b. To skip over orders that don't match the available ingredients

c. To count how many orders are in the queue

d. To create a copy of each order before returning it

Comparable (Revisited from Quiz 4)

(Use the code provided in the "Coupling / cohesion / encapsulation" section)

33. How does `__gt__` have us sort `Orders`?

a. First by burrito type, increasing, and then by spice level, increasing, if the burrito types are equal

b. First by burrito type, increasing, and then by spice level, decreasing, if the burrito types are equal

c. By burrito type only

d. By spice level only

34. If you tried to do `order1 < order2`, what would happen with the current implementation?

a. It would raise a `TypeError` because `__lt__` is not implemented

b. It would work correctly, taking the opposite of the boolean `order1 > order2`

c. It would work correctly, taking the opposite of the boolean `order1 > order2 or order1 == order2`

d. It would be `False` by default

35. Is there a potential inconsistency between `__eq__` and `__ne__`?

a. No inconsistency, because they always return the opposite of each other

b. Yes, because they can return different things from each other

c. Yes, because objects that were once equal can become unequal

d. No inconsistency, because they're unrelated operations

36. If we have two orders with the same customer name and burrito type but different spice levels, what will `order1 == order2` evaluate to?

a. `True`, because they have the same customer name and burrito type

b. `False`, because they have different spice levels

c. `TypeError`, because spice levels don't match

d. `None`, because equality is ambiguous

Recursion (Revisited from Quiz 4)

37. This recursive function has a subtle bug. What happens when we count the number of nodes in an empty tree? Note: The `TreeNode` class has attributes `self.left` and `self.right`, both of which are of type `Optional[TreeNode]`.

```
def count_nodes(node: Optional[TreeNode[T]]) -> int:
    if node.left is None and node.right is None:
        return 1
    else:
        return 1 + count_nodes(node.left) + count_nodes(node.right)
```

a. Returns 0

b. Infinite recursion

c. Raises AttributeError because 'NoneType' object has no attribute left

d. Raises TypeError because node's type doesn't allow it to be None

38. Which of the following will cause a stack overflow (infinite recursion)?

```
def process(n: int) -> int:
    if n == 0:
        return 1
    else:
        return n * process(n - 1)
```

a. `process(5)`

b. `process(0)`

c. `process(-3)`

d. `process(1)`

39. How many times is `count_vowels` called (including the initial call) when executed with `count_vowels("banana")`?

```
def count_vowels(s: str) -> int:
    if len(s) == 0:
        return 0
    elif s[0] in 'aeiou':
        return 1 + count_vowels(s[1:])
    else:
        return count_vowels(s[1:])
```

a. 0 times

b. 6 times

c. 7 times

d. 8 times

40. How many times is `mystery(3)` called when executing `mystery(5)`?

```
def mystery(n: int) -> int:
    if n <= 1:
        return n
    else:
        return mystery(n - 1) + mystery(n - 2)
```

a. 1 time

b. 2 times

c. 3 times

d. 4 times

Minimum Spanning Trees (Revisited from Quiz 4)

41. What happens if there is a graph with three edges that have the same weight, and adding any two of them would create a cycle? How many will be in the Minimum Spanning Tree built using Kruskal's Algorithm?

a. None of them will be included.

b. All three will be included.

c. Exactly one will be included.

d. Exactly two will be included.

42. What happens if a graph has some edges with positive weights, and some with negative weights?

a. Kruskal's algorithm fails

b. Kruskal's algorithm works fine

c. MSTs are undefined for graphs with negative weights

d. Negative weights automatically create cycles

43. Which of the following is NOT true about a Minimum Spanning Tree built using Kruskal's Algorithm?

a. It connects all nodes in the graph.

b. It has more edges than nodes.

c. It always includes the edge with the smallest weight in the graph, assuming there is not a tie for the smallest weight.

d. It may include the edge with the largest weight in the graph.

44. What happens in Kruskal's algorithm when we encounter an edge that would connect two nodes already in the same component?

a. We add the edge anyway to ensure connectivity.

b. We skip the edge because it would create a cycle.

c. We remove a previously added edge.

d. The algorithm terminates immediately.