Quiz 3 Review

Welcome back to CS 2100!

Prof. Rasika Bhalerao

Topics revisited from preious quizzes

- Classes (from Quiz 2)
  - Constructors, methods, attributes
  - `__str__()` and `__eq__()`
- Unit Testing (from Quiz 1)
  - `self.assertEqual()`, `self.assertTrue()`, and `self.assertRaises()`
  - Identifying test cases
- Using Objects (from Quiz 2)
  - State and aliasing

- Lists, sets, and dictionaries
  - List comprehension
  - Iterating through lists, sets, and dictionaries
  - Rules about contents of lists, sets, and dictionaries
  - Sorting and filtering
  - Binary operators (`|` , `-` , etc.)

New review topics

- Inheritance and bstract methods
  - `@abstractmethod`
  - Rules for instantiation
  - What subclasses inherit
  - Overwriting inherited methods
  - `super().`
- Properties
  - `@property` and `*.setter`
  - Attributes with `_` and `__`
- Iterator and Comparable
  - Iterable and Iterator protocols and interfaces
  - Comparable protocol
  - Checking for inconsistencies
  - Rules for using `<`, `>`, etc.

Abstract methods

We cannot instantiate a class that has an `@abstractmethod` (even inherited).

```python
from abc import ABC, abstractmethod

class Shape(ABC):
    @abstractmethod
    def get_area(self) -> float:
        pass

    @abstractmethod
    def get_perimeter(self) -> float:
        pass

shape = Shape()  # TypeError
```

- Doing so will raise a `TypeError`.
- To instantiate a class that inherits a `@abstractmethod`, overwrite it with a **concrete** (non-abstract) method.

Poll: What happens?

```python
from abc import ABC, abstractmethod

class Animal(ABC):
    @abstractmethod
    def make_sound(self) -> None:
        pass

class Dog(Animal):
    pass

dog = Dog()
```

1. A `Dog` object is created successfully
2. A `TypeError` is raised because `Dog` doesn't implement `make_sound()`
3. `Dog.make_sound()` returns `None`
4. A warning is printed but `dog` is created

Poll: Which is TRUE?

1. A class can only have one abstract method
2. You can instantiate an `ABC` if it has no abstract methods
3. Abstract methods cannot have parameters
4. `ABC` classes cannot have concrete (non-abstract) methods

Poll: What is output?

```python
from abc import ABC, abstractmethod

class Vehicle(ABC):
    @abstractmethod
    def start(self) -> str:
        return "Starting..."

class Car(Vehicle):
    def start(self) -> str:
        return super().start() + " car engine"

c = Car()
print(c.start())
```

A) A TypeError is raised

B) "Starting..."

C) "Starting... car engine"

D) "car engine"

Poll: How many abstract methods must a concrete (non-abstract) subclass overwrite?

1. At least one
2. Exactly one
3. All of them
4. None, they are optional

Inheritance

1. A *subclass* is a more specific version of a *superclass*.
2. The subclass *inherits* all methods and attributes from the superclass.

   -> Except those named with two underscores
3. The subclass can overwrite any inherited methods / attributes.
4. The subclass can add more methods / attributes.

Calling a superclass's method (or constructor)

```python
class Cat:
    def __init__(self, name: str):
        self.name = name
        self.food = ['tuna', 'chicken']

    def knead(self) -> None:
        print('Kneading')

    def eat(self, food: str) -> None:
        if food in self.food:
            print(f'Eating {food}')

class Lion(Cat):
    def __init__(self, name: str):
        super().__init__(name)
        self.food += ['zebra']

    def roar(self) -> None:
        print('Roaring')
```

- `eat()` method inherited from `Cat` works by default in `Lion`
- `self.food` is defined in `Cat`'s constructor, so we overwrite it with a new constructor in `Lion`...
    - one that executes `Cat`'s constructor first, and then adds `'zebra'` to `self.food` (which is inherited)

Poll: What is output?

```python
class Vehicle:
    def __init__(self, brand: str):
        self.brand = brand

class Car(Vehicle):
    def __init__(
      self, brand: str, model: str):
        self.model = model

c = Car("Toyota", "Camry")
print(c.brand)
```

1. `Toyota`
2. `None`
3. An `AttributeError` is raised
4. `Camry`

Poll: Which method call would return `"Rex makes a sound"`?

```python
class Animal:
    def __init__(self, name: str):
        self.name = name

    def speak(self) -> str:
        return f"{self.name} makes a sound"

class Dog(Animal):
    def __init__(self, name: str, breed: str):
        super().__init__(name)
        self.breed = breed

    def speak(self) -> str:
        return f"{self.name} barks"

d = Dog("Rex", "Labrador")
```

A) `d.speak()`

B) `super().speak()`

C) `Animal.speak(d)`

D) `d.Animal.speak()`

Attribute visibility

It is impossible to block an attribute from being accessed externally.

- `self.size` (attribute with no underscores): anyone can access
- `self._contents` (single underscore): nicely ask others to avoid using it
    - Externally accessible (`dataframe._contents`)
- `self.__password` (two underscores): even stronger suggestion to keep away
    - External name is mangled (`my_diary._Diary__password`)

Poll: Which are true?

```python
class BankAccount:
    def __init__(self, account_id: int, pin: str):
        self.balance = 0
        self._account_id = account_id
        self.__pin = pin
```

1. All three attributes are truly private and cannot be accessed from outside the class
2. `balance` and `_account_id` are publicly accessible
3. `_account_id` and `__pin` are not accessible from outside the class
4. Accessing `_account_id` from outside `BankAccount` is avoided by convention
5. Externally, `__pin` is name-mangled to `_BankAccount__pin`

Properties: `@property` and `*.setter`

- Create a property by putting the `@property` decorator above a method with the name for the property
  - Returns the value of the property (likely using `_` or `__` attributes)
- Give the property a "setter" using another method with the same name, with the decorator `@age.setter`
  - Takes the property's new value as an arg
  - Updates any internal attributes

```python
class Person:
    def __init__(self, age: int):
        self._age = age

    @property
    def age(self) -> int:
        return self._age

    @age.setter
    def age(self, new_age: int) -> None:
        if new_age >= 0:
            self._age = new_age


mini: Person = Person(10)
mini.age = 11
print(mini.age) # 11
```

Poll: Which code snippet will work?

```python
class Temperature:
    def __init__(self, celsius: float):
        self._celsius = celsius

    @property
    def fahrenheit(self) -> float:
        return self._celsius * 9/5 + 32

    @fahrenheit.setter
    def fahrenheit(
            self, value: float) -> None:
        self._celsius = (value - 32) * 5/9
```

1.

```python
temp = Temperature(0)
print(temp.fahrenheit(0))
temp.fahrenheit(32)
```

2.

```python
temp = Temperature(0)
print(temp.fahrenheit)
temp.fahrenheit = 32
```

3.

```python
temp = Temperature(0)
print(temp.get_fahrenheit())
temp.set_fahrenheit(32)
```

4.

```python
temp = Temperature(0)
print(temp._fahrenheit)
temp._fahrenheit = 32
```

Iterable / Iterator

	Iterable	Iterator
Protocol's required methods	`__iter__(self) -> Iterator[T]` : returns an iterator	`__next__(self) -> T` : returns the next element or raises `StopIteration` `__iter__(self) -> Iterator[T]` : returns itself
`abc` interface's required methods	`__iter__(self) -> Iterator[T]` (same as protocol)	`__next__(self) -> T` (same as protocol) not `__iter__(self) -> Iterator[T]` because it's aleady there

Exercise: let's write a class `Sarcasm`, which is like a `str`, but when we iterate over it, it capitalizes a random half of the letters

```python
import random
from collections.abc import Iterable, Iterator

class Sarcasm(Iterable[str]):
    def __init__(self, text: str):
        self.text = text

    def __iter__(self) -> Iterator[str]:
        return SarcasmIterator(self.text)

class SarcasmIterator(Iterator[str]):
    def __init__(self, text: str):
        self.remaining_text = text

    def __next__(self) -> str:
        if len(self.remaining_text) == 0:
            raise StopIteration
        next_char = self.remaining_text[0]
        next_char = next_char.lower() if random.randint(0, 1) == 0 else next_char.upper()
        self.remaining_text = self.remaining_text[1:]
        return next_char

print(''.join(letter for letter in Sarcasm('hi rasika')))
```

Comparable

- `__eq__(self, other: object) -> bool`: equals `==`
- `__ne__(self, other: object) -> bool`: not equals `!=`
- `__lt__(self, other: object) -> bool`: less than `<`
- `__le__(self, other: object) -> bool`: less than or equal to `<=`
- `__gt__(self, other: object) -> bool`: greater than `>`
- `__ge__(self, other: object) -> bool`: greater than or equal to `>=`

Don't need all six (which is why there's no interface)

Common: Implement `__eq__()` and one ordering method like `__lt__()`

`a < b` calls `a.__lt__(b)` or `not a.__ge__(b)` or `not (a.__gt__(b) or a == b)`

Poll: How can we check for inconsistencies between comparison methods?

1. If they use the same attributes for comparison, then they must be consistent.
2. If `__eq__()` says A is equal to B, and `__gt__()` says A is greater than B, then they are inconsistent.
3. If `__le__()` says A is less than or equal to B, and `__ge__()` says A is greater than or equal to B, then they are inconsistent.
4. If all six comparison methods are implemented, then they must be inconsistent.

Let's go through Practice Quiz 3!

Poll:

1. What is your main takeaway from today?

2. What would you like to revisit next time?