Quiz 4 Review

Welcome back to CS 2100!

Prof. Rasika Bhalerao

Revisited topics

Using Objects

- State and aliasing
- `None` and `Optional`

Stakeholder-value matrices (revisited from Quiz 2)

- Selecting stakeholders
- Selecting values

Correlation (revisited from Quiz 2)

- Definition of correlation
- Pearson's correlation coefficient

New Quiz 4 Topics

Coupling / cohesion / encapsulation

- Identifying / mitigating coupling between classes
- Identifying / mitigating lack of cohesion
- Enhancing encapsulation a class

Iterator

- Iterable and Iterator protocols and interfaces

Comparable

- Comparable protocol
- Checking for inconsistencies
- Rules for using `<`, `>`, etc.

Decorator, Strategy, Observer, and Data Pull Design Patterns

- Why and how to make a function decorator
- When to use the Strategy pattern (and how)
- When to use Observer versus Data Pull (and how)

Cohesion / coupling / encapsulation

Cohesion: how closely related the parts of a unit are (good)

- Single Responsibility Principle: each unit should have exactly one responsibility
- Function: has a single, well-defined job
- Class: methods are very closely related

Coupling: how dependent separate units are (bad)

- Often, that means that one class is too dependent on another, and any changes to the other class will result in "ripple effects" on it.

Encapsulation: how hidden complex details are (good)

- shields clients from unnecessary implementation details
- gives us more flexibility to change implementations without telling the client

Poll: Which ones lead to good cohesion?

1. Put all methods in one big class instead of several small classes
2. Make sure each variable represents exactly one piece of information
3. Use helper methods to split complex tasks into multiple simple tasks
4. Write code that runs efficiently

Poll: How can we avoid coupling?

1. Make it so that changing a class's attributes doesn't require us to to update any code in another class
2. Don't use any built-in Python types
3. Make it so a class doesn't instantiate or control instances of another class
4. Write thorough tests

Poll: Which ones improve encapsulation?

1. Using underscores in variable names to indicate they shouldn't be directly accessed
2. Using appropriate variable names (other than the underscores)
3. Using properties so we can control how attributes are modified
4. Writing thorough documentation

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.

Decorator, Strategy, Observer, and Data Pull Design Patterns

Design pattern	When to use it	How to use it
(Function) Decorator	When we want to modify a function's behavior	Write a function that takes and returns a function, use its `@function_name` as the decorator
Strategy	When we want to choose an algorithm at runtime	Have each algorithm option implement an interface, and use that interface in the main program
Observer	When many objects request the same data over and over	Make the data producer keep a list of consumers to be updated every time the data changes
Data Pull	When data gets updated often, but requests for that data happen less often	Make each consumer call the same method in the data producer

Practice Quiz 4

Poll:

1. What is your main takeaway from today?
2. What would you like to revisit next time?