

# Quiz 4 Review

**Welcome back to CS 2100!**

**Prof. Rasika Bhalerao**

# Revisited topics

- Inheritance and abstract 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.
- Stakeholder-value matrices
  - Selecting stakeholders
  - Selecting values
- Correlation
  - 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

## **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

# 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 <code>@function_name</code> 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?**