ACIT 2515

Object Oriented Programming in Python – Week 6

Instructor: Mike Mulder

Agenda – Week 6

- Topics:
  - Python Class Review
  - Built In Methods
  - Design Patterns and Template Method
- Quiz 5
- Midterm Preview
- Lab 6 – In Class
- Assignment 1 Questions
  - Assignment Due Sunday, June 19th

Anatomy of a Python Class

```
# Example of a Point Class
class Point:

    MIN_X = 0
    MAX_X = 10000
    MIN_Y = 0
    MAX_Y = 10000

    def __init__(self, x, y):
        self.move(x, y)

    def move(self, x, y):
        Point.validate_xy(x, y)
        self._x = x
        self._y = y

    def reset(self):
        self.move(0, 0)

    def print_details(self):
        print(self._x, self._y)

    @staticmethod
    def validate_xy(x, y):
        if x < Point.MIN_X or x > Point.MAX_X:
            raise ValueError("Bad X")
        if y < Point.MIN_Y or y > Point.MAX_Y:
            raise ValueError("Bad Y")
```

Class

Class Variables

Initializer/ Constructor

Instance Variables

Methods

Static Method

```
# Example of using the Point Class
point1 = Point(50, 75)
point2 = Point(5, 10)
point1.move(35, 57)
point2.reset()
point1.print_details()
point2.print_details()
print(Point.MAX_X)
print(Point.MAX_Y)
Point.validate_xy(10001, 10001)
```

Creating Objects

Invoking Instance Methods

Referencing Class Variables

Invoking Static Methods

```
Output:
35 57
0 0
10000
10000
<Raises Exception Here>
```

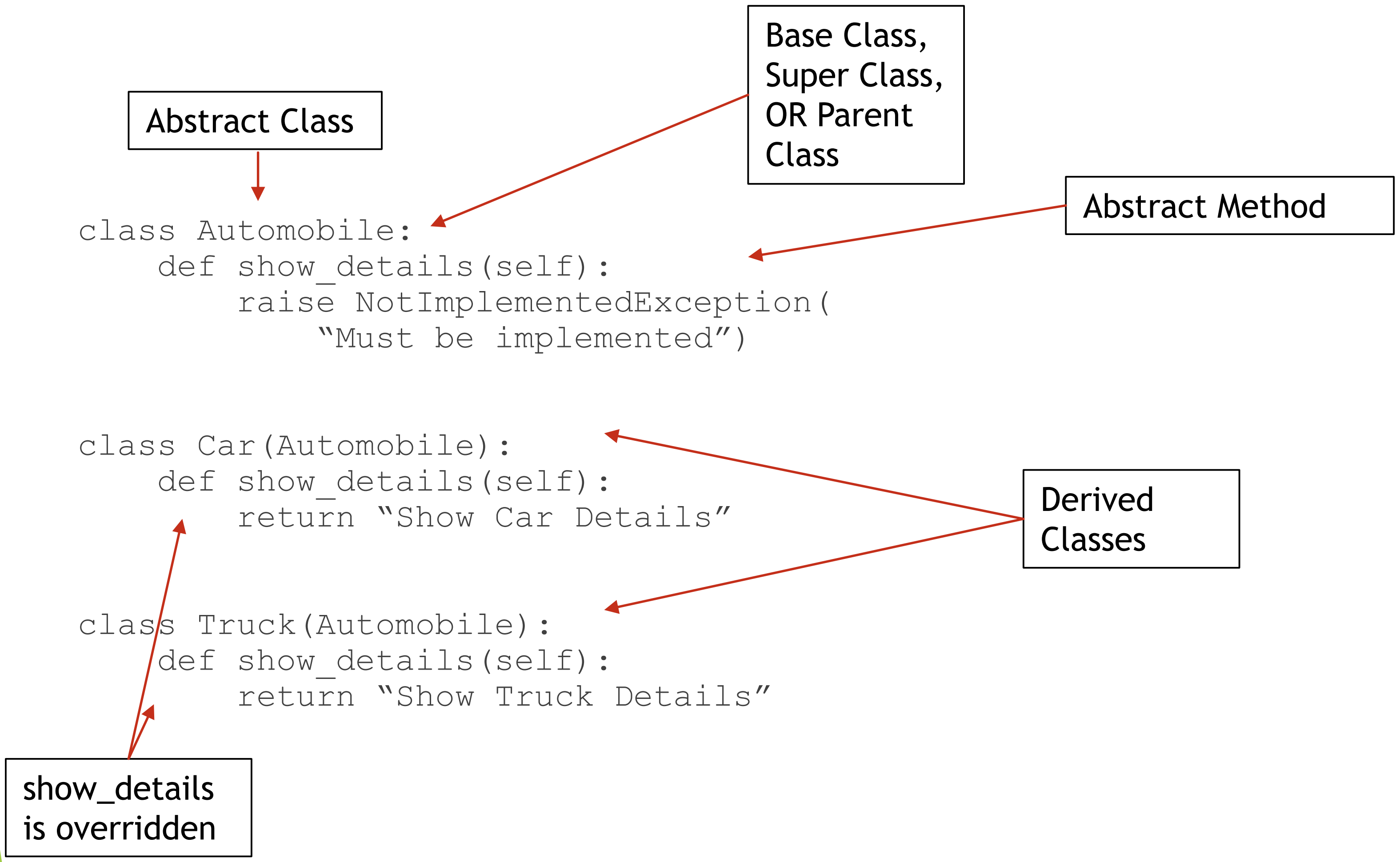
__init__ is the initializer of an object in Python

self is a reference to the object that the instance method is being invoked on.

@staticmethod is a decorator to indicate a method is static on a class

Anatomy of a Python Class

Inheritance and Polymorphism


Abstract Class
Base Class, Super Class, OR Parent Class
Abstract Method
Derived Classes
show_details is overridden

```
class Automobile:
    def show_details(self):
        raise NotImplementedException(
            “Must be implemented”)


class Car(Automobile):
    def show_details(self):
        return “Show Car Details”


class Truck(Automobile):
    def show_details(self):
        return “Show Truck Details”
```

Anatomy of a Python Class

Multiple Inheritance

```
class AddressContact:
    def get_address(self):
        return self._address


class EmailContact:
    def get_email(self):
        return self._email


class Contact(AddressContact, EmailContact):
    def show_details(self):
        return self.get_address() + " " + self.get_email()
```

Base Classes, Super Classes, OR Parent Classes

Derived Class

Anatomy of a Python Class

Overriding Constructor

Base Class,
Super Class, OR
Parent Class

```
class User:
    def __init__(self, first_name, last_name):
        self._first_name = first_name
        self._last_name = last_name
```

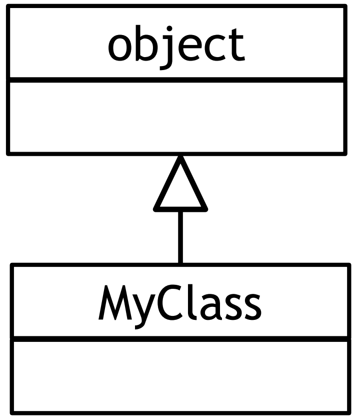
Derived
Class

```
class Student(User):
    def __init__(self, first_name, last_name, date_enrolled):
        super().__init__(first_name, last_name)
        self._date_enrolled = date_enrolled
```

super() returns the object as an instance of the parent class. It can be used in other methods too, not just __init__.

Built-In Methods in Python Classes

- All Python Classes are implicitly derived from the **object** class. It is the top-level parent of all classes in Python.
- So "class Point" is effectively equivalent to "class Point(object)"
  - In Python 3, we just specify the class definition as "class Point". The inheritance relationship to the top-level object is automated created.
- The top-level object defines a number of useful "built-in" methods.



What is an example of one of these methods that we have already used in every class we've created?

__init__

Built-In Methods in Python Classes

Some other "built-in" methods:

- __new__ - Lets you customize how an object is created (rarely used)
- __init__ - Lets you initialize your newly created object (i.e., the constructor)
- __call__ - Automatically called if the object is called like a function

```
point1 = Point(5, 10)
point1()  # Triggers the __call__ method
```

- __str__ - "Official" output for describing the object – human readable

```
str(point1) # Returns the output from __str__ for point1
```

- __repr__ - Debugging output for an object – extra developer info

```
repr(point1) # Returns the output from __repr__ for point1
```

We can override these methods in our own classes.

Built-In Methods - Example

What is output in each of the following cases:

```
point1 = Point(10, 20)

print(point1)

print(str(point1))

print(repr(point1))
```

With overridden __str__ and __repr__

```
def __str__(self):
    return "My Output Str: %f,%f" %
            (self._x, self._y)

def __repr__(self):
    return "My Output Repr: %f,%f" %
            (self._x, self._y)

My Output Str: 10.000000,20.000000
My Output Str: 10.000000,20.000000
My Output Repr: 10.000000,20.000000
```

With default __str__ and __repr__

```
<point.Point object at
0x0000026EC6973438>
<point.Point object at
0x0000026EC6973438>
<point.Point object at
0x0000026EC6973438>
```

Design Patterns

What is a Design Pattern?

A design pattern is a general repeatable solution to a commonly occurring problem in software design

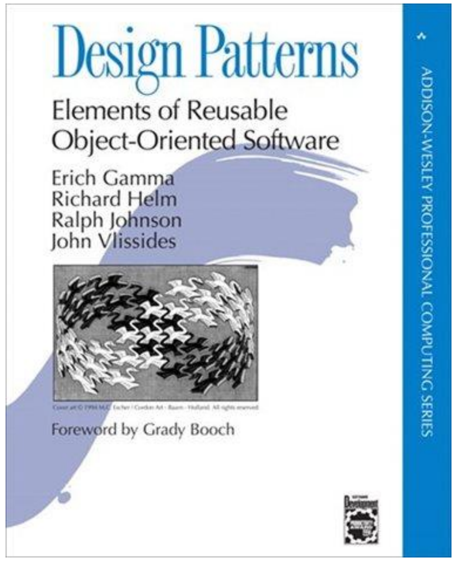
It is not a finished design. It is a template for how to solve a problem that can be applied in many different situations.

Note that a design pattern is NOT code.
It is a reusable design that can be implemented in code.

Where Did They Come From?

Design Patterns: Elements of Reusable Object Oriented Software

Authored by the Gang of Four (GoF) – Gamma, Helm, Johnson and Vlissides – in 1994


Design Patterns
Elements of Reusable Object-Oriented Software
Erich Gamma
Richard Helm
Ralph Johnson
John Vlissides
Foreword by Grady Booch
ADDISON-WESLEY PROFESSIONAL COMPUTING SERIES

Categories of Design Patterns?

Creational Design Patterns

- Patterns about Class instantiation

Structural Design Patterns

- Patterns about Class and Object composition

Behavioral Design Patterns

- Patterns about Class's objects communication

Source: https://sourcemaking.com/design_patterns

Creational Design Patterns

Abstract Factory	Creates an instance of several families of classes
Builder	Separates object construction from its representation
Factory Method	Creates an instance of several derived classes
Object Pool	Avoid expensive acquisition and release of resources by recycling objects that are no longer in use
Prototype	A fully initialized instance to be copied or cloned
Singleton	A class of which only a single instance can exist

Source: https://sourcemaking.com/design_patterns

Structural Design Patterns

Adapter	Match interfaces of difference classes
Bridge	Separates an object’s interface from its implementation
Composite	A tree structure of simple and composite objects
Decorator	Add responsibilities to objects dynamically
Façade	A single class that represents an entire subsystem
Flyweight	A fine-grained instance used for efficient sharing (i.e., cache)
Private Class Data	Restricts access/mutator data access
Proxy	An object representing another object

Source: https://sourcemaking.com/design_patterns

Behavioral Design Patterns

Chain of Responsibility	A way of passing a request between a chain of objects
Command	Encapsulate a command request as an object
Interpreter	A way to include language elements in a program
Iterator	Sequentially access the elements of a collection
Mediator	Defines simplified communication between classes
Memento	Capture and restore an object’s internal state
Null Object	Designed to act as a default value of an object
Observer	A way of notifying change to a number of classes
State	Alter an object’s behavior when it’s state changes
Strategy	Encapsulates an algorithm inside a class
Template Method	Defer the exact steps of an algorithm to a subclass
Visitor	Defines a new operation to a class without change

Source: https://sourcemaking.com/design_patterns

Design Patterns – Labs 4 and 6

Private Class Data

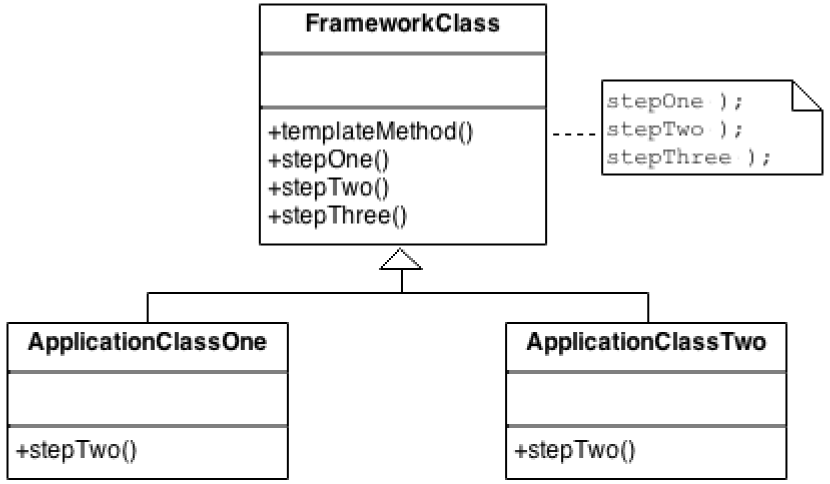
- Lab 4 (Last Two Weeks)
- Created a data class for the **RepairStats**
- Classes just for holding the data
- All data is private
- Use getters to access the data
- In this case, the output from the methods is more readable as the attributes are named (rather than list indices).

Template Method

- Lab 6 (This Week)
- AbstractRobot, RandomRobot and MyRobot classes
- Abstract methods in the base (parent) class (AbstractRobot) must be implemented by the derived (child) classes (RandomRobot and MyRobot)
- The base (parent) class provides the template for the implementation of the derived (child) classes through the abstract methods.

Design Pattern We’re Using Today

Template Method Design Pattern



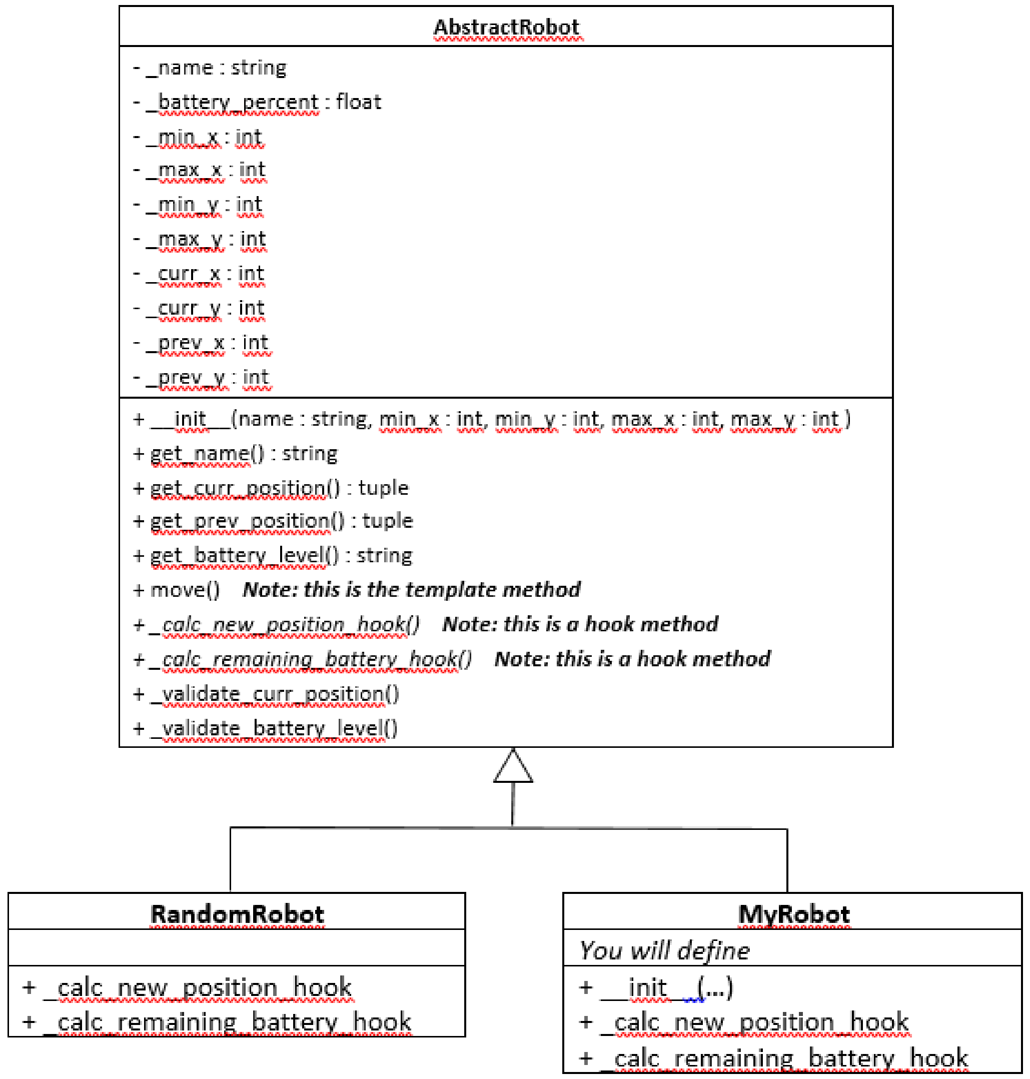
The parent FrameworkClass has a templateMethod() that calls one or more hook methods. The hook methods may or may not be abstract.

These hook methods can be overridden to implement behavior specific (or unique) to the child classes.

Source: https://sourcemaking.com/design_patterns/template_method

Design Pattern We're Using Today

Template Method Design Pattern


AbstractRobot
- _name : string
- _battery_percent : float
- _min_x : int
- _max_x : int
- _min_y : int
- _max_y : int
- _curr_x : int
- _curr_y : int
- _prev_x : int
- _prev_y : int
+ __init__(name : string, min_x : int, min_y : int, max_x : int, max_y : int)
+ get_name() : string
+ get_curr_position() : tuple
+ get_prev_position() : tuple
+ get_battery_level() : string
+ move() Note: this is the template method
+ _calc_new_position_hook() Note: this is a hook method
+ _calc_remaining_battery_hook() Note: this is a hook method
+ _validate_curr_position()
+ _validate_battery_level()
RandomRobot
+ _calc_new_position_hook
+ _calc_remaining_battery_hook
MyRobot
You will define
+ __init__(...)
+ _calc_new_position_hook
+ _calc_remaining_battery_hook

In Lab 6 Today, the *move* method of our parent AbstractRobot class is the template method. We have abstract _calc_new_position_hook and _calc_remaining_battery_hook methods as hooks to define the specific behavior ofr the robot in terms of movement and battery life.

Benefits of Design Patterns

- Speed up development with proven and tested development paradigms
- Design and code reuse, prevents common issues through proven designs
- Better code readability
- General solutions, documented in a format that isn’t tied to the specifics of a particular problem
- Better developer communication – using common names for software patterns

Design Patterns in this Couse

- You do not need to memorize all the design patterns
- You should be aware they exist, and be familiar with those we use in the Labs and Assignments
- So far, we have used:
  - Private Data Classes
  - Template Method (today)
  - Model View Controller (MVC) (when we cover Graphic User Interfaces)

Criticisms of Design Patterns

- Targets the wrong problem
  - Originally targeted at C++
  - Programming languages should provide built-in solutions
- Lacks formal foundations
  - Study of design patterns was adhoc
- Leads to inefficient solutions
  - Design pattern is an attempt to standardize already accepted best practices
  - May lead to over-duplication of code and inefficient implementations
- Not different from other forms of Abstraction (i.e., good design in general)
- Overuse of design patterns

  Applying designs patterns when not appropriate (i.e., if someone has a hammer then everything looks like a nail)

SOLID Principles

- Single responsibility principle
  - a class should have only a single responsibility (i.e. changes to only one part of the software's specification should be able to affect the specification of the class).
- Open/closed principle
  - "software entities … should be open for extension, but closed for modification."
- Liskov substitution principle
  - "objects in a program should be replaceable with instances of their subtypes without altering the correctness of that program." See also design by contract.
- Interface segregation principle
  - "many client-specific interfaces are better than one general-purpose interface."
- Dependency inversion principle
  - one should "depend upon abstractions, [not] concretions."

Single Responsibility Principle

- Every module or class should have responsibility over a single part of the functionality provided by the software, and that responsibility should be entirely encapsulated by the class
- All its services should be narrowly aligned with that responsibility
- A class should have only one reason to change

What might be some examples of violations of the Single Responsibility Principle?

Classic Example: MVC – separate model (i.e. data) class, separate controller class and separate presentation class/template. This is a separation of concerns into different classes.

OOP Benefits (i.e, The Why)

- **Classes** – The building blocks of our software – testable and reusable.
- **Abstraction** – Good design practices (i.e., public interface, aka API).
- **Encapsulation** – Enabling change by hiding our implementation behind the public interface.
- **Inheritance** – Code reuse through parent/child relationships.
- **Composition** – Code reuse by composing objects of other objects.
- **Polymorphism** – Enables code reuse by allowing some methods to change to deal with specialization.
- **Design Patterns** – Standardization which enables faster development, fewer defects (i.e., proven designs) and developer communication

The goal of OOP was to promote standardization and reuse of software components. It emphasizes good design (decomposition in to objects), well-defined APIs (i.e., the public interface) and design/code re-use.

Quiz 5

- You have up to 25 minutes to complete the quiz.
- Open Book/Laptop, but do your own work.
- We will go over the answers afterwards (i.e., during the lecture).

Midterm Preview - Details

Tuesday, June 21st at 6pm

Online – Virtual Classroom and D2L Quiz

You have a total of 3 hours to complete

Will be conducted via a D2L Quiz and Dropbox

There are 2 parts to the exam

Midterm Preview - Written

Part 1 - Written (25 marks)

- Five Topics plus Code Review:
  - Classes and Objects
    - Definitions and Anatomy of a Python Class (including inheritance and polymorphism)
  - Unit Testing (including test fixtures and TDD)
  - UML Diagrams (class notation, composition, aggregation and inheritance relationships)
    - Including writing and instantiating a simple class from a UML class diagram
  - Object Oriented Principles (abstraction, encapsulation, inheritance/composition and polymorphism)
  - Design Patterns, definition, benefits, criticisms and Template Method
  - Code Review – Identify style violations and errors in a given class definition
- This part of the exam will be a combination of short answer, coding and multiple choice type questions.

Midterm Preview - Coding

Part 2 – Coding (25 marks)

- You will be asked to create 2-4 simple Python classes with composition, aggregation and/or inheritance relationships. You will also create a Python script that instantiates the classes, generates some specific output and answer some questions on the design.
  - Best practices for naming, documentation, constants, parameter validation etc. is expected in your code.
- Your completed code must be uploaded to a dropbox in D2L before the end of the exam time.

Midterm Preview – Study Aids

Written Part

- Use the lecture slides and quizzes 1-5 to study.
- Midterm questions will be of a similar format to the quizzes.

Coding Part

- Some practice questions have been posted to D2L.
- There will be no unit test required but a test script is required to produce specific output to the console.
- You may be asked one or two questions analyzing the given design and/or implementation in terms of the 4 pillars.

Lab 6

- Due in-class today
- There is code on D2L (Week 6) as a starting point
- Demo to you Instructor in Discord and submit the code when you are done to get marks

If you finish early, work on your Assignment 1. If you have UML diagrams I can review on Discord (after I've finished the lab demos).