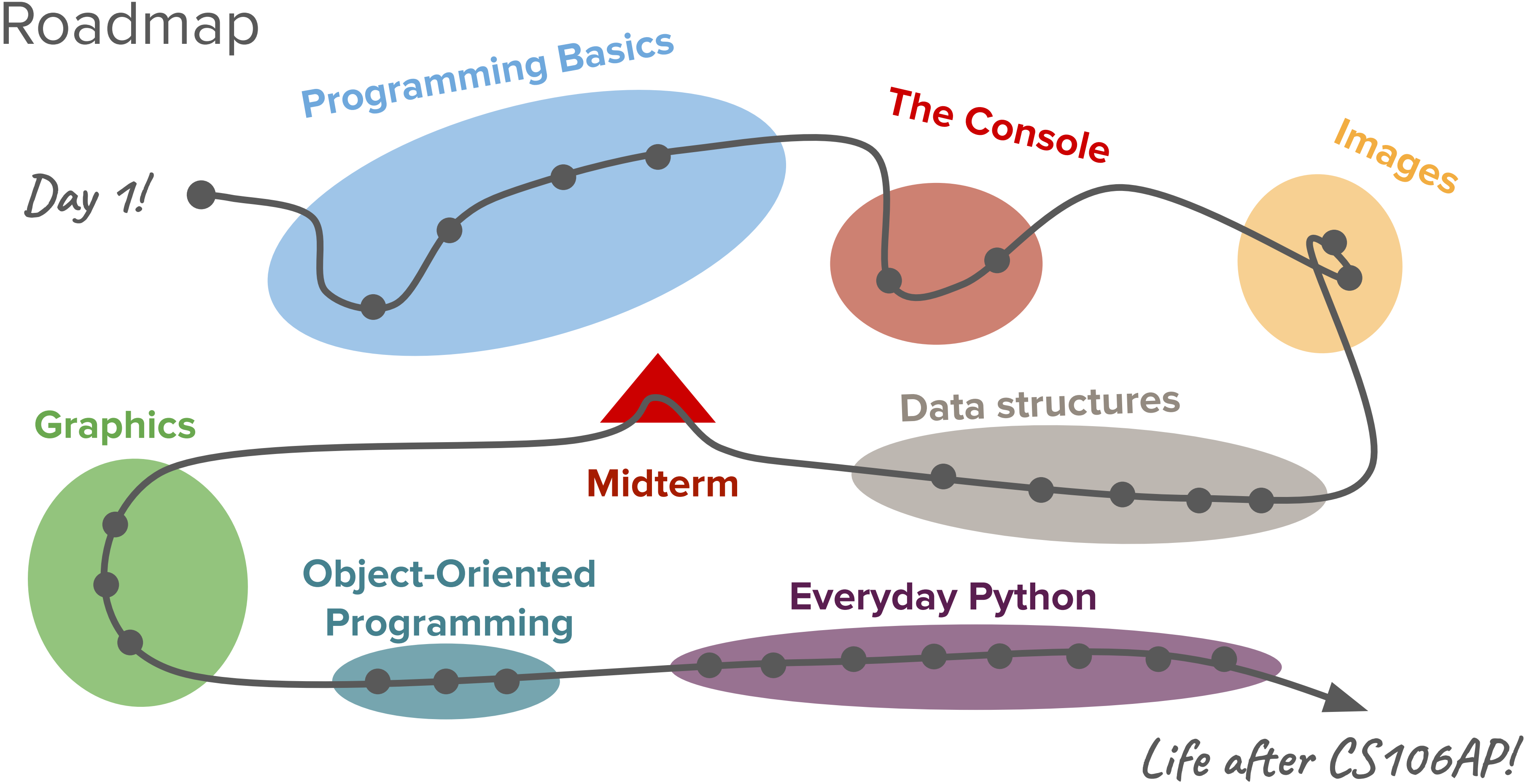
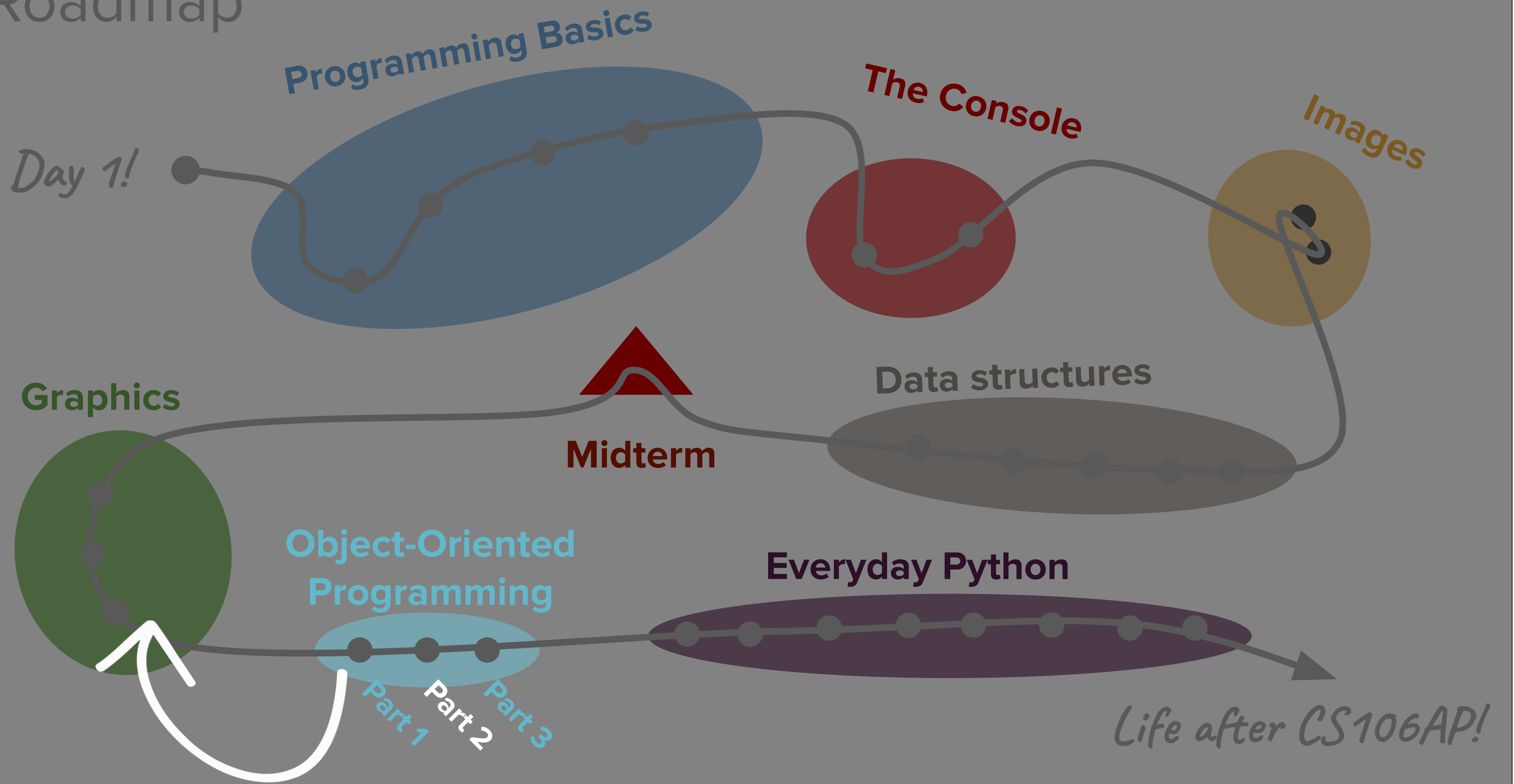
# Event-Driven Programming and Abstraction

CS106AP Lecture 21

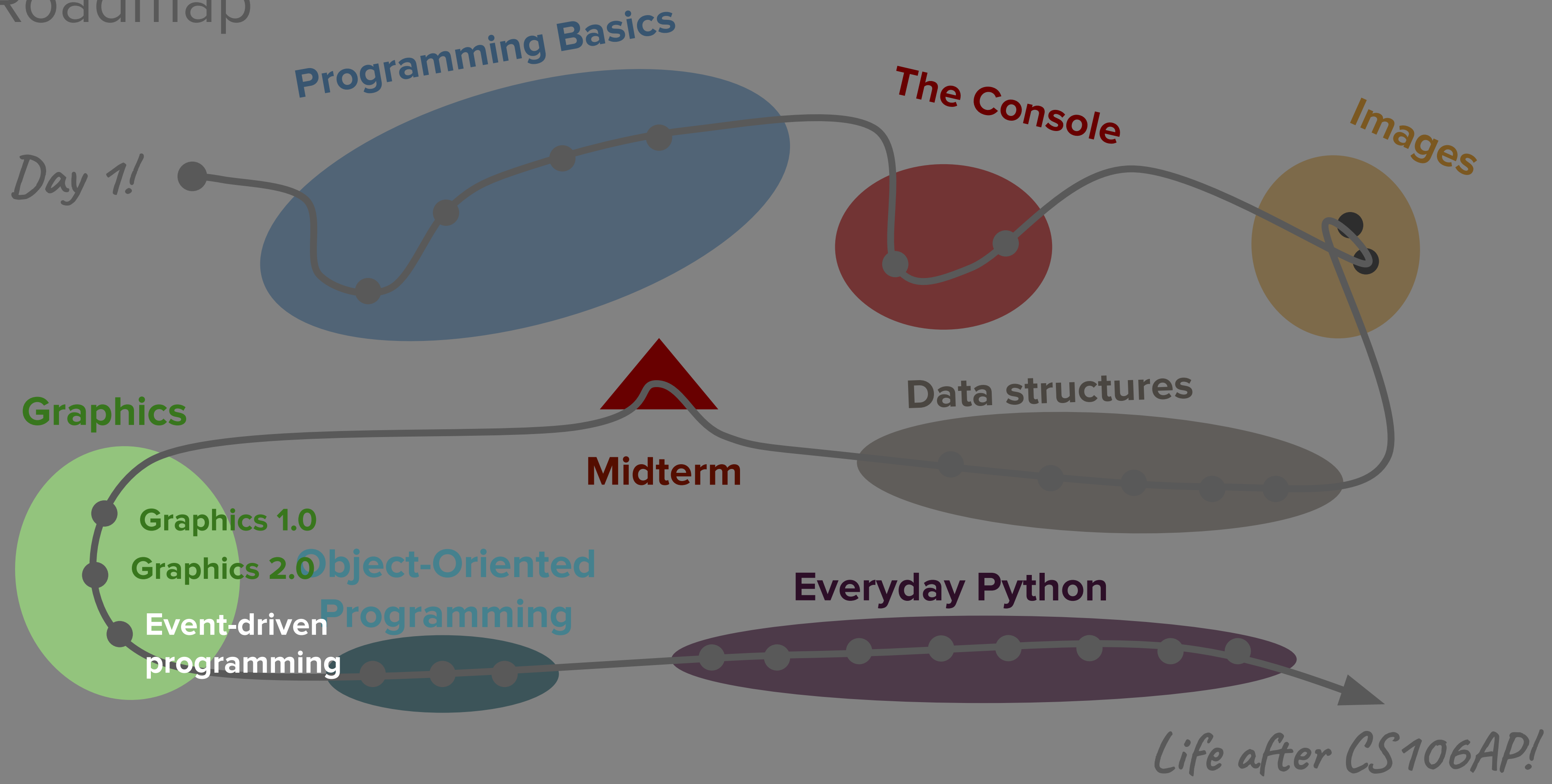
# Roadmap


Day 1!
Programming Basics
The Console
Images
Data structures
Midterm
Graphics
Object-Oriented Programming
Everyday Python
Life after CS106AP!

# Roadmap


Day 1!
Programming Basics
The Console
Images
Graphics
Midterm
Data structures
Object-Oriented Programming
Everyday Python
Part 1
Part 2
Part 3
Life after CS106AP!

# Roadmap


Programming Basics
The Console
Images
Day 1!
Data structures
Graphics
Midterm
Graphics 1.0
Graphics 2.0
Object-Oriented Programming
Everyday Python
Event-driven programming
Life after CS106AP!

# Today's questions

How can we write programs that respond to user actions?

Why do we use classes when writing code for other people to use?

# Today's topics

1. Review
2. Event-driven programming
3. Classes and abstraction
4. What's next?

# Review

# Encapsulation

# Encapsulation is bundling info into one nice package!

- Integration
  - All the smaller parts add up to create the entire functionality
  - Similar to top-down decomposition

# Encapsulation is bundling info into one nice package!

- Integration

- Modular development
  - You can separate different types of tasks and know where different information/functionality should be.
  - Easier for testing and debugging!

# Encapsulation is bundling info into one nice package!

- Integration

- Modular development

- Instance variables (attributes)
  - Knowledge (data) for a specific class **stays inside** that class.
  - That information is **easier to access** across methods **within** that class.
  - If you need to access the information **outside** the class, there's a **predefined structure** for doing so.

# Encapsulation is bundling info into one nice package!

- Integration
- Modular development
- Instance variables (attributes)
  - Knowledge (data) for a specific class **stays inside** that class.
  - That information is **easier to access** across methods **within** that class.
  - If you need to access the information **outside** the class, there’s a **predefined structure** for doing so.

More later today!

# Bubbles.py

[more bubbles!]

How do we write programs that respond to user actions?

# How do we write programs that respond to user actions?

*Event-driven programming!*

# The event listener model

Your code

```
def main():

    ...

    ...

def your_mouse_listener():

    ...
```

# The event listener model

Your code

```
def main():
    ...
    ...
def your_mouse_listener():
    ...
```

Definition

**mouse listener function**

A function that occurs immediately when a user triggers a particular mouse event

# The event listener model

Your code

```
def main():

    ...

    ...

def your_mouse_listener():

    ...
```

*Definition*

**mouse listener function**
A function that occurs immediately when a user triggers a particular **mouse event**

*clicking, moving, dragging*

# The event listener model

Your code

```
def main():

    ...

    ...

def your_mouse_listener():

    ...
```

# The event listener model

Your code

```
def main():

    ...

    ...

def your_mouse_listener():

    ...
```

# The event listener model

Your code

```
def main():

    ...

    ...

def your_mouse_listener():

    ...
```

# The event listener model

Your code

```
def main():

    ...

    ...

def your_mouse_listener():

    ...
```

# The event listener model

Your code

```
def main():
    ...
    ...
def your_mouse_listener():
    ...
```

*The function happens immediately, no matter where you are in your program!*

# Creating a mouse listener

1. Write a mouse listener function (handler)

```
def mouse_listener_handler(event):
    ...
```

# Creating a mouse listener

1. Write a mouse listener function (handler)

```
def mouse_listener_handler(event):
    ...
```

It must take in an **event** for campy to recognize it as a valid mouse listener.

# Creating a mouse listener

1. Write a mouse listener function (handler)

```
def mouse_listener_handler(event):

    ...
```

**`event`** gives us access to information about the mouse event (e.g. x, y coordinates of the click).

# Creating a mouse listener

1. Write a mouse listener function (handler)

```
def mouse_listener_handler(event):
    ...
```

2. Use the corresponding campy **`onmouseevent()`** function to set up your mouse listener

```
onmouseclicked(mouse_listener_handler)
```

# Creating a mouse listener

1. Write a mouse listener function (handler)

```
def mouse_listener_handler(event):
    ...
```

2. Use the corresponding campy **`onmouseevent()`** function to set up your mouse listener

```
onmouseclicked(mouse_listener_handler)
```

Pass in your mouse listener function as the argument

# Creating a mouse listener

1. Write a mouse listener function (handler)

```
def mouse_listener_handler(event):
    ...
```

2. Use the corresponding campy **`onmouseevent()`** function to set up your mouse listener

```
onmouseclicked(mouse_listener_handler)
```

Don't include parentheses after the function name!

# Bubbles.py

[mouse listener demo]

# Creating a mouse listener

1. Write a mouse listener function (handler)

```
def mouse_listener_handler(event):
    ...
```

2. Use the corresponding campy **`onmouseevent()`** function to set up your mouse listener

```
onmouseclicked(mouse_listener_handler)
```

# Mouse Listeners and Classes

1. Write a mouse listener function (handler)

```
def mouse_listener_handler(self, event):
    ...
```

2. Use the corresponding campy **onmouseevent()** function to set up your mouse listener

```
onmouseclicked(self.mouse_listener_handler)
```

Don't include parentheses after the function name!

# Why do we use classes?

Yesterday!

- For ourselves
  - Grouping related data and the functions that act on it
  - Modular code development (isolation of particular tasks)

- For others
  - We hide the implementation details of our code so others don’t need to worry about them.
  - They can just use the class, like we do for SimpleImage.

# Why do we use classes?

- For ourselves
  - Grouping related data and the functions that act on it
  - Modular code development (isolation of particular tasks)

Today!

- For others
  - We hide the implementation details of our code so others don’t need to worry about them.
  - They can just use the class, like we do for SimpleImage.

# Why do we use classes in code meant for others?

# Why do we use classes in code meant for others?

*Abstraction!*

## Definition

**abstraction**

Hiding implementation details of a class from the clients of that class

other programmers!

# Clients and Interfaces

- Classes—or really any code we write (modules, libraries, etc.)—can be thought of from two perspectives.

Slide adapted from Jerry Cain

# Clients and Interfaces

- Classes—or really any code we write (modules, libraries, etc.)—can be thought of from two perspectives.
  - The code for the class itself is called the **implementation**.
    - For example, all the code we've written inside the **`BubbleGraphics`** class

*Slide adapted from Jerry Cain*

# Clients and Interfaces

- Classes—or really any code we write (modules, libraries, etc.)—can be thought of from two perspectives.
  - The code for the class itself is called the implementation.
    - For example, all the code we've written inside the **`BubbleGraphics`** class
  - Any code that uses a class in any way is called the **client**
    - For example, the **`animate_bubble_pop()`** or **`animate_many_bubbles()`** functions we wrote today

*Slide adapted from Jerry Cain*

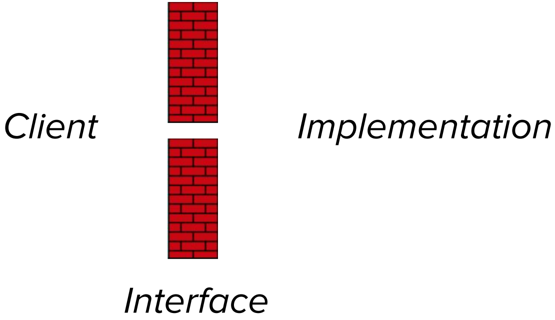
# Clients and Interfaces

- Classes—or really any code we write (modules, libraries, etc.)—can be thought of from two perspectives.
- The point at which the client and implementation meet and communicate is known as the **interface**, which serves as both a barrier and a communication channel

*Slide adapted from Jerry Cain*

# Clients and Interfaces

- Classes—or really any code we write (modules, libraries, etc.)—can be thought of from two perspectives.
- The point at which the client and implementation meet and communicate is known as the **interface**, which serves as both a barrier and a communication channel



*Client*

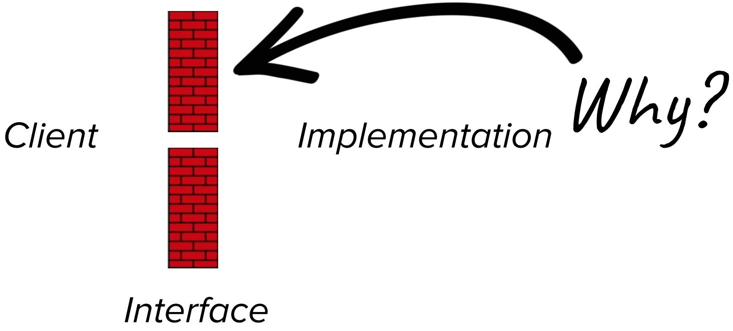
*Implementation*

*Interface*

*Slide adapted from Jerry Cain*

# Clients and Interfaces

- Classes—or really any code we write (modules, libraries, etc.)—can be thought of from two perspectives.
- The point at which the client and implementation meet and communicate is known as the **interface**, which serves as both a barrier and a communication channel



Slide adapted from Jerry Cain

# Information Hiding

- One of the central principles of modern software design is that each level of abstraction should hide as much complexity as possible from the layers that depend on it. This principle is called **information hiding**.

*Slide adapted from Jerry Cain*

# Information Hiding

- One of the central principles of modern software design is that each level of abstraction should hide as much complexity as possible from the layers that depend on it. This principle is called **information hiding**.
- When you **use** a function, it is more important to know what the function does than to understand exactly how it works.

*Slide adapted from Jerry Cain*

# Information Hiding

- One of the central principles of modern software design is that each level of abstraction should hide as much complexity as possible from the layers that depend on it. This principle is called **information hiding**.
- When you **use** a function, it is more important to know what the function does than to understand exactly how it works.
  - The underlying details are of interest only to the programmer who implements the function.

*Slide adapted from Jerry Cain*

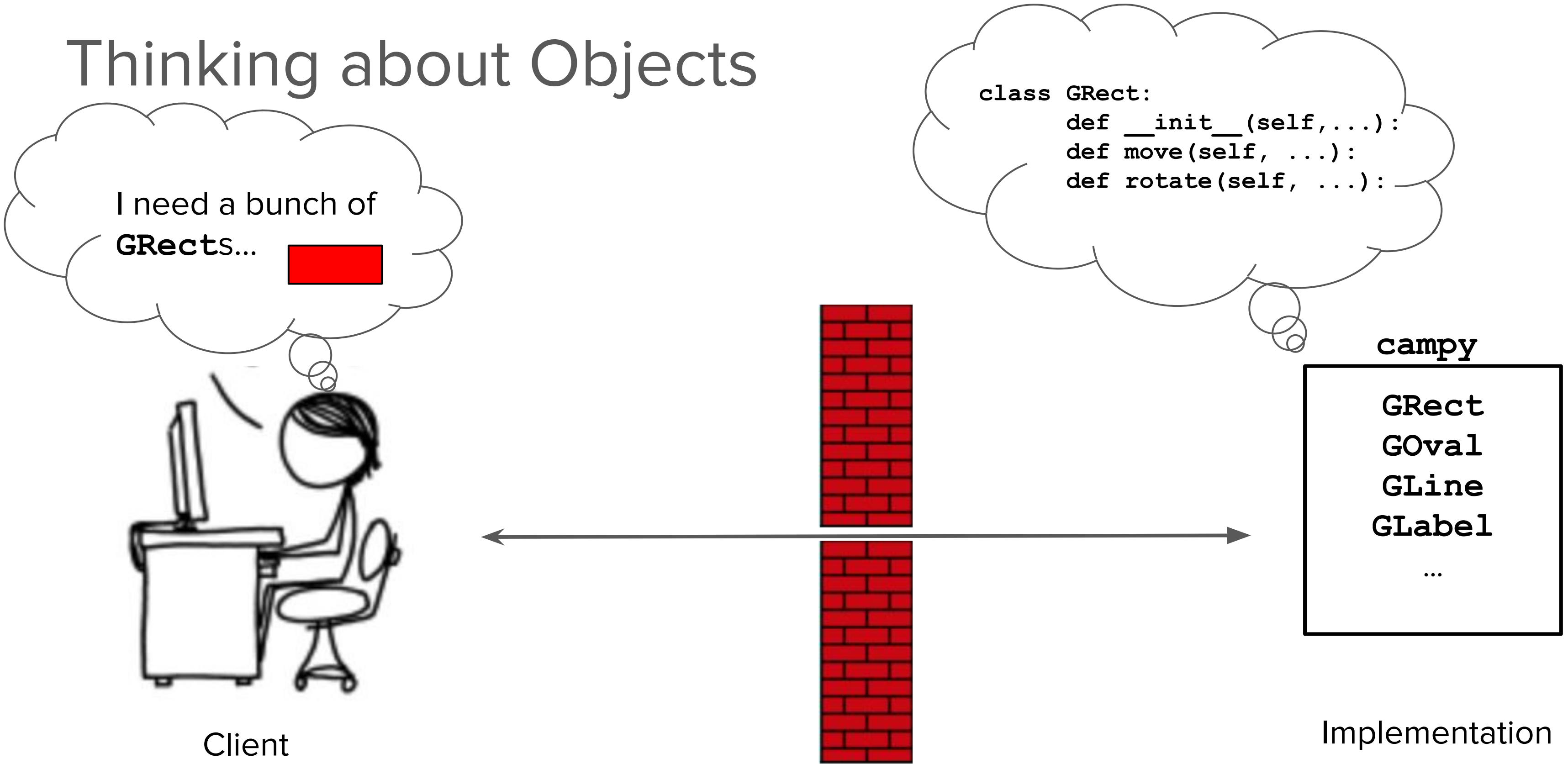
# Information Hiding

- One of the central principles of modern software design is that each level of abstraction should hide as much complexity as possible from the layers that depend on it. This principle is called **information hiding**.

- When you **use** a function, it is more important to know what the function does than to understand exactly how it works.
  - The underlying details are of interest only to the programmer who implements the function.
  - Clients who use that function as a tool can usually ignore the implementation altogether.

*Slide adapted from Jerry Cain*

# Thinking about Objects



I need a bunch of **`GRect`**s...

```
class GRect:
      def __init__(self,...):
      def move(self, ...):
      def rotate(self, ...):
```

**`campy`**

```
GRect
GOval
GLine
GLabel
...
```

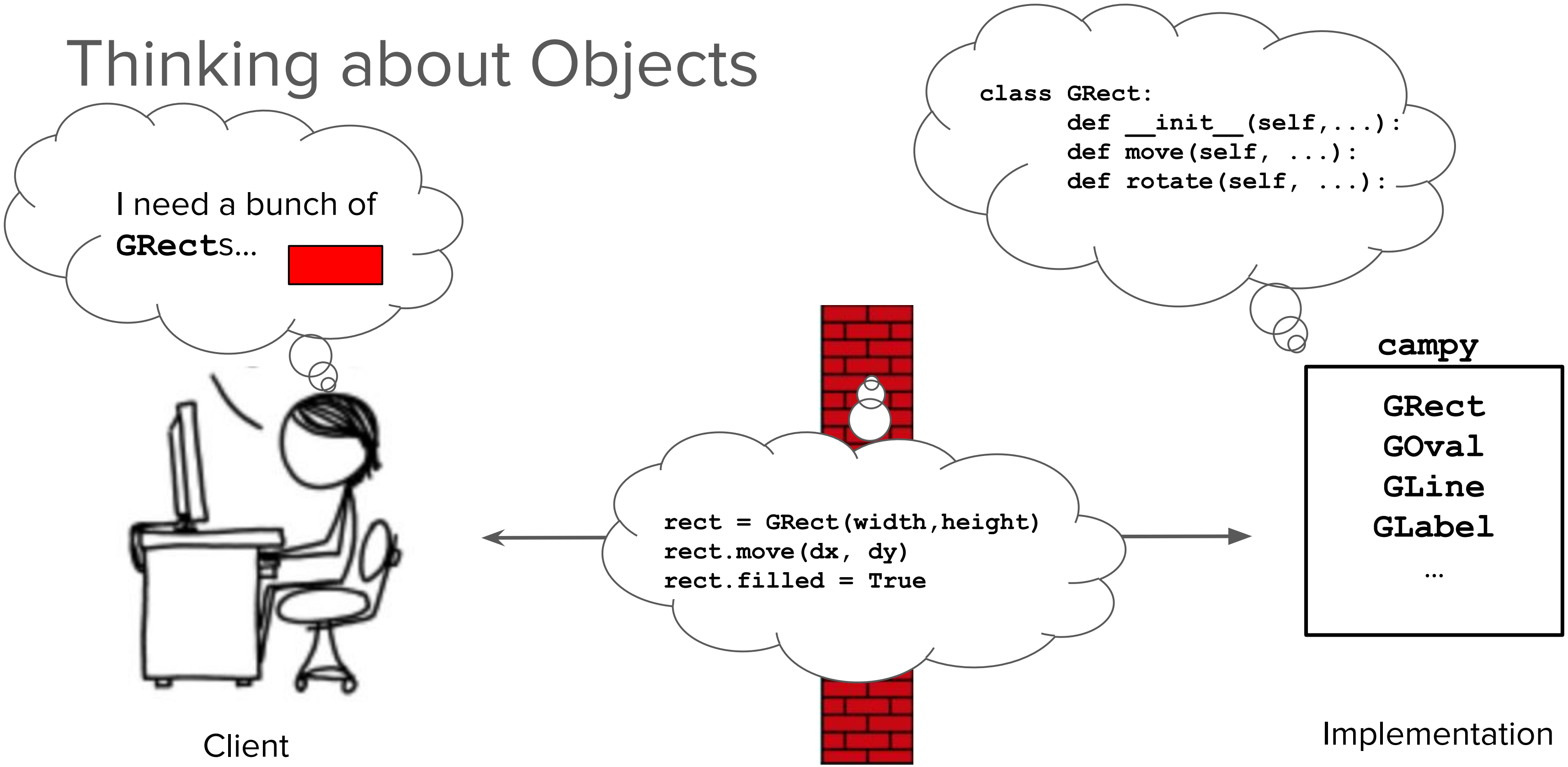
Client

Implementation

Abstraction boundary (interface)

*Diagram adapted from Jerry Cain and image credit to Randall Munroe at xkcd.xom*

# Thinking about Objects



I need a bunch of **GRect**s...

```
class GRect:
      def __init__(self,...):
      def move(self, ...):
      def rotate(self, ...):
```

```
rect = GRect(width,height)
rect.move(dx, dy)
rect.filled = True
```

**campy**

**GRect**
**GOval**
**GLine**
**GLabel**
...

Client

Implementation

Abstraction boundary (interface)

*Diagram adapted from Jerry Cain and image credit to Randall Munroe at xkcd.xom*

# Abstraction protects the data stored in an object

- Getters and setters are the interface to the data
    - These functions provide clients with a specific, limited way of accessing the data.
    - If clients could change the data in any way they wanted, things could get really messy.

# Abstraction protects the data stored in an object

- Getters and setters are the interface to the data
  - These functions provide clients with a specific, limited way of accessing the data
  - If clients could change the data in any way they wanted, things could get really messy.

- Clients don’t have to worry about constraints on the data
  - The implementation will handle that for them behind-the-scenes!
  - E.g. A **`PynstaUser`** shouldn’t be able to add a friend they’re already friends with.

# Abstraction protects the data stored in an object

- Getters and setters are the interface to the data
- Clients don’t have to worry about constraints on the data

*An example!*

PayPal

P yPal

# PyPal.py

[abstraction demo]

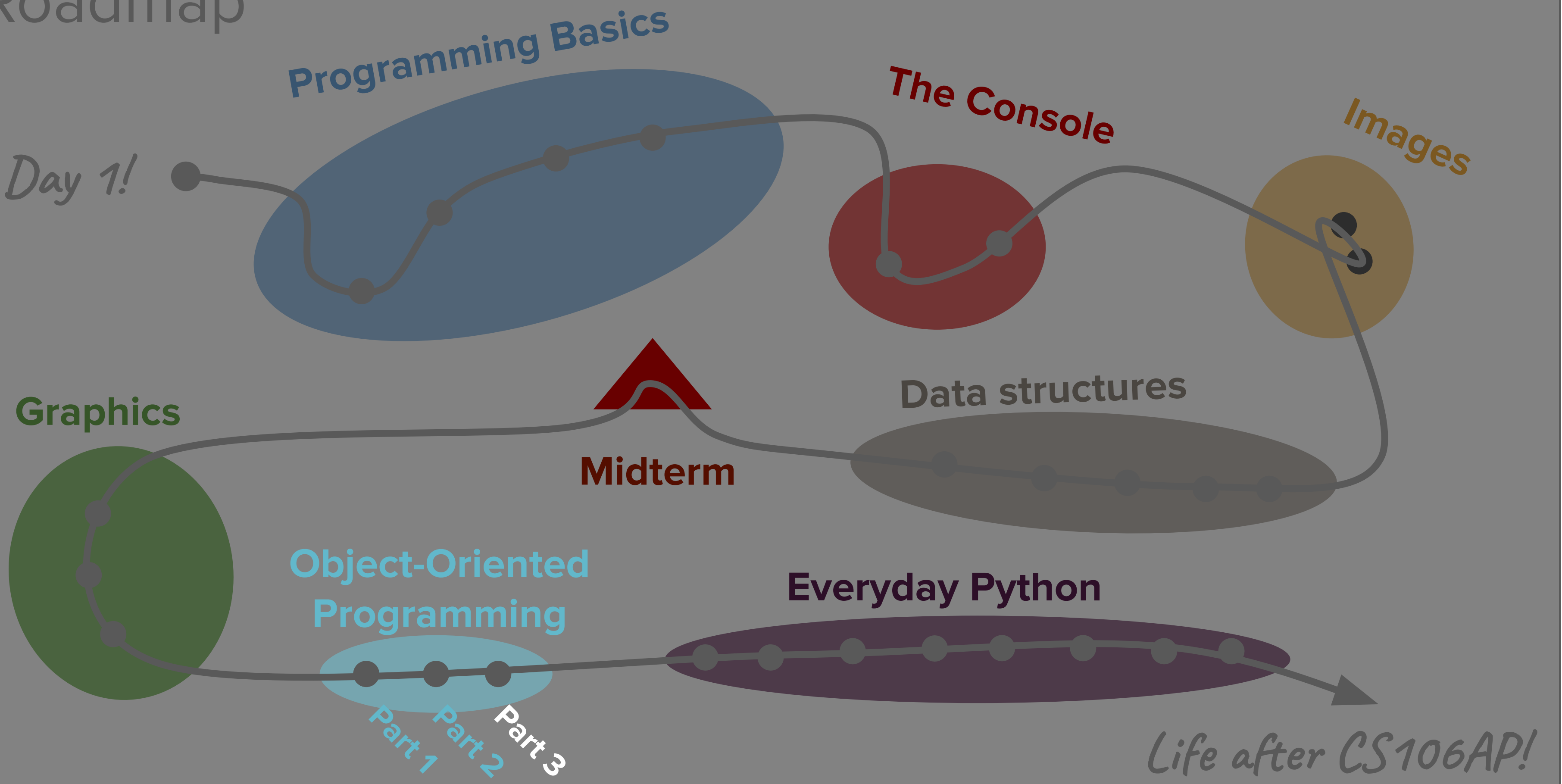
# What’s next?

# Putting it all together!

- How we can leverage encapsulation and abstraction to build complex graphical programs that interact with users

- Using all of the skills we’ve learned so far to code a fun game!

# Roadmap


Day 1!
Programming Basics
The Console
Images
Graphics
Midterm
Data structures
Object-Oriented Programming
Part 1
Part 2
Part 3
Everyday Python
Life after CS106AP!