



# Object-Oriented Programming (OOP)

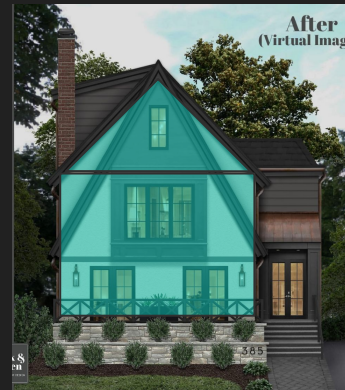
## Part 2: Classes and Methods

# Announcements

- **LS13: Intro to Object-Oriented Programming** due today at 11:59pm
- **EX06: Dictionary Unit Tests** released today, due Monday (3/30) at 11:59pm

# Review: **Classes** and **objects**

- Think of a **class** as a blueprint/  
template
  - Defines attributes and behaviors its  
objects will have
- An **object** is an *instance* of a class
  - E.g., if the class is the blueprint, the  
object is the house!
  - Has all the specified attributes and  
behaviors
  - Different objects share these  
attributes and behaviors, but are  
distinct!



# Modeling an Instagram profile with a `class`

declaring a new data type!

```
class Profile:
    username: str
    bio: str
    followers: int
    following: int
    private: bool
```

declaring attributes  
(every Instagram profile has these!)

```
def __init__(self):
    self.username = "usr9"
    self.bio = ""
    self.followers = 0
    self.following = 0
    self.private = False
```

initializing attributes  
(what are the default values?)

```
my_prof: Profile = Profile()
my_prof.username = "comp110fan"
print(my_prof.private)
```

# Memory diagram

```
1 class Profile:
2     username: str
3     bio: str
4     followers: int
5     following: int
6     private: bool
7
8     def __init__(self):
9         self.username = ""
10        self.bio = ""
11        self.followers = 0
12        self.following = 0
13        self.private = False
14
15
16 my_prof: Profile = Profile()
17 your_prof: Profile = Profile()
18 your_prof.username = "unccompsci"
19 my_prof.username = "unc.csx1"
20
21 print(my_prof.username)
```

# Returning to our goal: modeling an Instagram profile with code

What data should we keep track of?

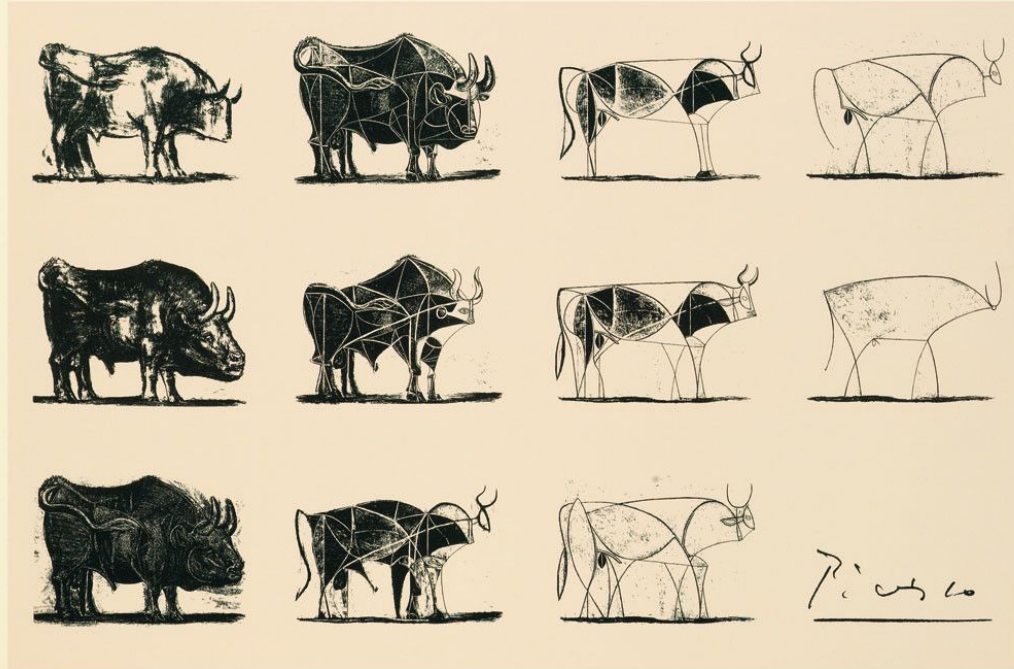
```
username: str = "unc.csxl"  
bio: str = "UNC CS Experience Labs"  
posts: int = 37  
followers: int = 322  
following: int = 123  
private: bool = False
```

What behaviors would be useful?

- View # followers or following
- Write or update a bio
- (Un)follow an account
- Make an account private/public

How can we write code to enable these actions for any Instagram account?

# What does Picasso's "Bull" progression show?



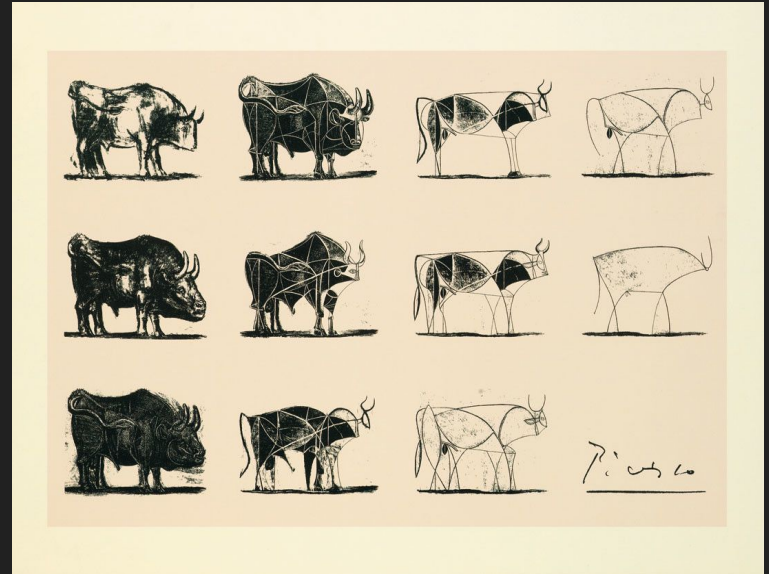
Pablo Picasso. Bull (1945). A Lithographic Progression.

# Abstraction: whittling down to the essentials

## Real-world example: Flights

What information do you need when you're preparing for (or actively on) a flight?

- ❑ ALL of the flight details?
  - ❑ E.g., how the pilot flies the plane
- or,*
- ❑ Only the ones that are essential for you to know?
  - ❑ Departure and arrival times/cities, your seat assignment, plans after landing



Pablo Picasso. Bull (1945).  
A Lithographic Progression.

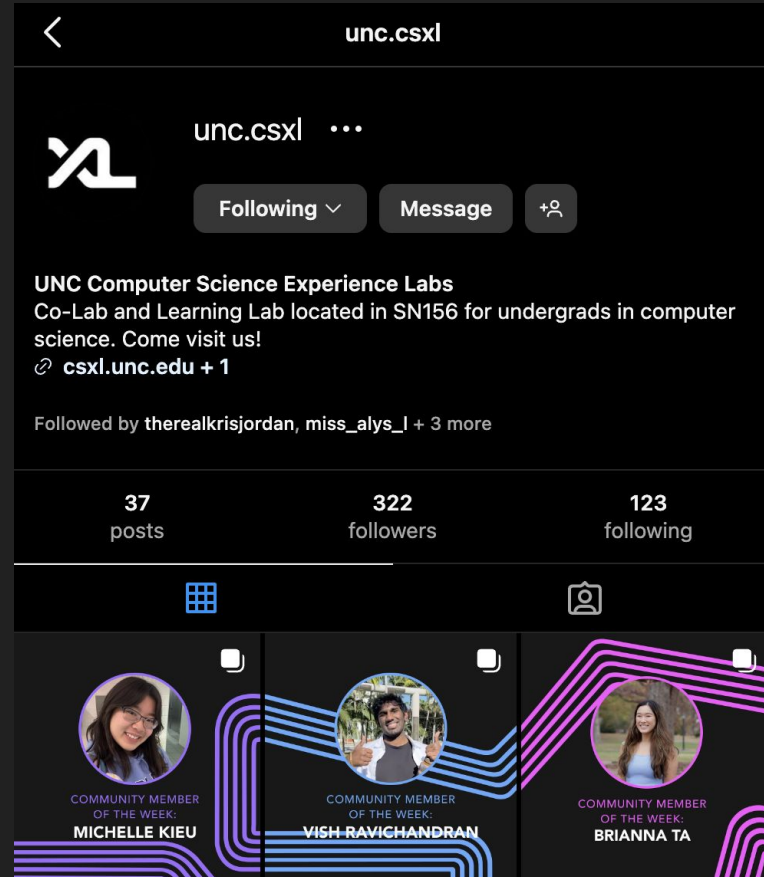
# Abstraction: whittling down to the essentials

## Today's example: Instagram Profiles

When you:

- ❑ Follow someone
- ❑ Make your account private
- ❑ Post a new photo

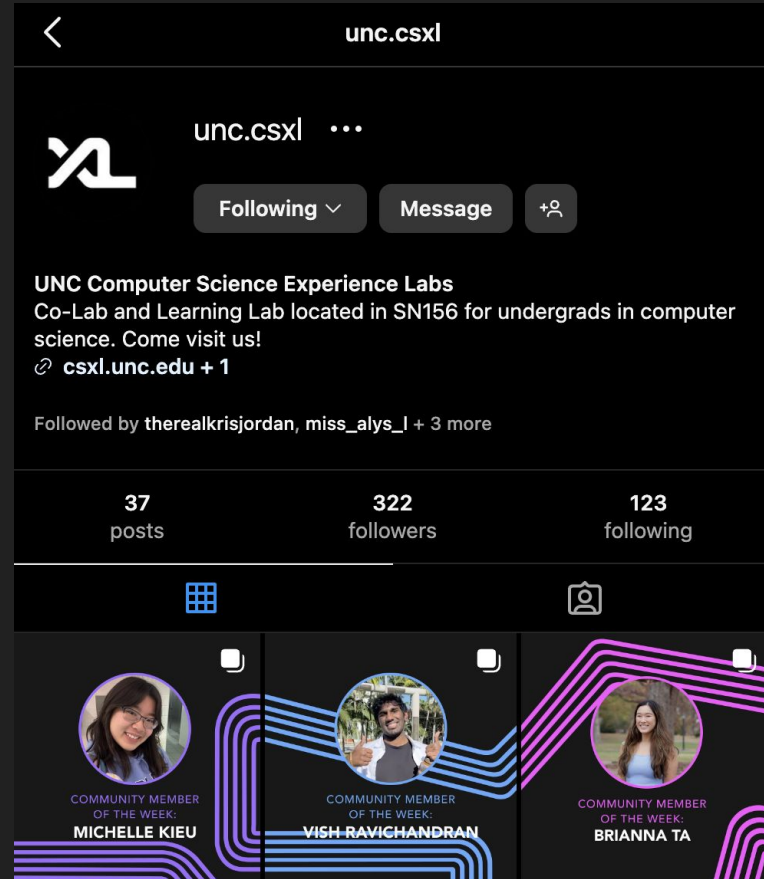
Do you think about what's happening behind the scenes (in Meta's code)?



# Objects are a data abstraction

All objects have:

1. An **internal representation**
  - a. Data attributes
2. An **interface** for interacting with the object
  - a. Interface defines behaviors but *hides implementation* (the details!)
  - b. **Methods**: Functions defined within a class
    - i. `self` is the first parameter



# Methods: defined in the *class*, called on *objects*

```
1 class Profile:
2     username: str
3     followers: list[str]
4     following: list[str]
5
6     def __init__(self, handle: str):
7         self.username = handle
8         self.followers = []
9         self.following = []
10
11     # Method definitions
12     def follow(self, username: str) -> None:
13         self.following.append(username)
14
15     def following_count(self) -> int:
16         return len(self.following)
17
18 my_prof: Profile = Profile("comp110fan") # Calls __init__()
19
20 my_prof.follow("hack110_unc")
21 print(my_prof.following_count())
```

Method definitions  
(first parameter is `self`!)

Method call  
<object>.<method>( <non-self arguments> )

# Memory diagram

```
1 class Profile:
2     username: str
3     followers: list[str]
4     following: list[str]
5
6     def __init__(self, handle: str):
7         self.username = handle
8         self.followers = []
9         self.following = []
10
11     # Method definitions
12     def follow(self, username: str) -> None:
13         self.following.append(username)
14
15     def following_count(self) -> int:
16         return len(self.following)
17
18 my_prof: Profile = Profile("comp110fan")
19
20 my_prof.follow("hack110_unc")
21 print(my_prof.following_count())
```

# Code writing

```
1 class Point:
2     x: float
3     y: float
4
5     def __init__(self, x: float, y: float):
6         self.x = x
7         self.y = y
8
9     def dist_from_origin(self) -> float:
10        return (self.x**2 + self.y**2) ** 0.5
11
12    def translate_x(self, dx: float) -> None:
13        self.x += dx
14
15
16 p0: Point = Point(10.0, 0.0)
17 p0.translate_x(-5.0)
18 print(p0.dist_from_origin())
```

Following line 18, write additional lines of code that:

1. Declares an additional variable of type Point and initializes it to a new Point object with coordinates (1.0, 2.0)
2. Call the translate\_x method on your Point object, passing an argument of 1.0.
3. Print the value returned by calling the dist\_from\_origin method on your Point object.

What would the printed output be?  
(This is great additional practice to try diagramming!)

Want more practice?

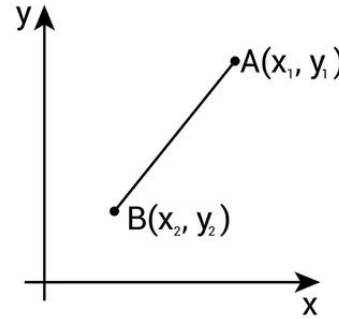
# Memory Diagram

```
1 class Point:
2     x: float
3     y: float
4
5     def __init__(self, x: float, y: float):
6         self.x = x
7         self.y = y
8
9     def dist_from_origin(self) -> float:
10        return (self.x**2 + self.y**2) ** 0.5
11
12    def translate_x(self, dx: float) -> None:
13        self.x += dx
14
15
16 p0: Point = Point(10.0, 0.0)
17 p0.translate_x(-5.0)
18 print(p0.dist_from_origin())
```

# Class and method writing

- Write a class called **Coordinate**
- It should have two attributes:
  - **x: float** and **y: float**
- Write a **constructor** that takes three parameters:
  - **self**, **x (float)** and **y (float)**
- Write a method called **get\_dist** that takes as parameters **self** and **other** (another **Coordinate** object). The method should return the distance between the two **Coordinate** objects (use the equation above!).

## Distance Formula



$$d = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$