Welcome class to **Week 02**!

Great job so far!

	Mon	7/13	▶ Part 1: Classes and Objects ▶ Part 2: Motivation, Player Types **Guest Lecturer:** Erica Goodwin	Coding practice Open work time
	Tue	7/14	▼ Exploring Pygame lecture slides	Open work time
			[Assignment Due] Project Milestone #2: Basic Game Design (Tue 7/14)	
Week 2	Wed	7/15	▶ Part 1: Inclusive Design, UDL ▶ Part 2: Designing Interactive Systems to Support Children's Critical and Reflective Engagement with AI **Guest Lecturer:** Erica Goodwin, Aayushi Dangol	Coding practice: Pygame warmup Open work time
	Thu	7/16	▶ Collapsing Qubits: A Quantum Themed Card Game **Guest Lecturer:** David Gonzalez-Maldonado	Coding practice: Mini game (choose 1) Open work time
	Fri	7/17	(Students) — Flash Talk and Peer Review Workshop	Field Trip: Weston Game Lab
			[Assignment Due] Project Milestone #3: Final Game Design & MVP Description (Sat 7/18)	

Week 2	Mon	7/13	▶ Part 1: Classes and Objects ▶ Part 2: Motivation, Player Types **Guest Lecturer:** Erica Goodwin	Coding practice Open work time
	Tue	7/14	▼ Exploring Pygame lecture slides	Open work time very short coding practice
			[Assignment Due] Project Milestone #2: Basic Game Design (Tue 7/14)	
	Wed	7/15	▶ Part 1: Inclusive Design, UDL ▶ Part 2: Designing Interactive Systems to Support Children's Critical and Reflective Engagement with AI **Guest Lecturer:** Erica Goodwin, Aayushi Dangol	Coding practice: Pygame warmup Open work time
	Thu	7/16	▶ Collapsing Qubits: A Quantum Themed Card Game **Guest Lecturer:** David Gonzalez-Maldonado	Coding practice: Mini game (choose 1) Open work time 2 hrs: tour & play
	Fri	7/17	(Students) — Flash Talk and Peer Review Workshop 5% of your course grade !!	Field Trip: Weston Game Lab
			[Assignment Due] Project Milestone #3: Final Game Design & MVP Description (Sat 7/18)	

Overview, reminders:

03 Guest Lecturers!!

(attendance)

Daily coding practices

Pygame, Minigame Design

- Milestone #2: Tuesday
- Flash Talk: Friday (5%)
- Game Lab Field Trip

Assignment descriptions are up on course websites!!

Today:

Mon	7/13	▶ Part 1: Classes and Objects ▶ Part 2: Motivation, Player Types **Guest Lecturer: Erica Goodwin**	**Coding practice** Open work time

9 - 10: Part I - Classes and Objects

10 - 10:15: Break

10:15 - 11:30: Part II - Motivation, Player Types

(Guest Lecturer Erica Goodwin)

13 - 15: Common Mistakes, Coding Practice, open work time

Today:

Mon	7/13	▶ Part 1: Classes and Objects ▶ Part 2: Motivation, Player Types **Guest Lecturer: Erica Goodwin**	**Coding practice** Open work time

9 - 10: Part I - Classes and Objects

10 - 10:15: Break

10:15 - 11:30: Part II - Motivation, Player Types

(Guest Lecturer Erica Goodwin)

13 - 15: Common Mistakes, Coding Practice, open work time

15-17: Office Hour

Python Objects

Which components make up these worlds & maps?

Object Oriented

- A program (or a game platform) is made up of many cooperating components
- We can organize these components into different “objects” that each do some work

(i.e. animals, background, character, user data...)

Object

- An Object is a bit of self-contained Code and Data
- A key aspect of the Object approach is to break the problem into smaller understandable parts (divide and conquer)
- We have been using objects all along: String Objects, Integer Objects, Dictionary Objects, List Objects...

Definitions

- Class - a template (i.e. character, trees, animals,...)
- Method or Message - A defined capability of a class
- Field or attribute - A bit of data in a class (i.e. names…)
- Object or Instance - A particular instance of a class

  (i.e. Character1, Character2, Krixi, Valhein,...)

A Sample Class

class is a reserved word that defines a template for making objects

Each PartyAnimal object has a bit of code

Tell the an object to run the party() code within it

```
class PartyAnimal:

  def __init__(self):
    self.x = 0

  def party(self) :
    self.x = self.x + 1
    print("So far",self.x)

an = PartyAnimal()

an.party()
an.party()
an.party()
```

In `an = PartyAnimal()`, the special `__init__` method is called to create the object.

Each `an.party()` will call PartyAnimal's party() method, and increment the class variable `x`

```
class PartyAnimal:

  def __init(self)__:
    self.x = 0

  def party(self) :
    self.x = self.x + 1
    print("So far",self.x)

an = PartyAnimal()

an.party()
an.party()
an.party()
```

Running the code:

```
class PartyAnimal:

  def __init__(self):
    self.x = 0

  def party(self) :
    self.x = self.x + 1
    print("So far",self.x)

an = PartyAnimal()

an.party()
an.party()
an.party()
```

Running the code:
So far 1
So far 2
So far 3

Object Lifecycle

- Objects are created, used, and discarded
- We have special blocks of code (methods) that get called
  - At the moment of creation (constructor) __init__
  - At the moment of destruction (destructor) __del__
- Constructors (the code in the `__init__`) are used a lot
- Destructors (`__del__`) are not used often

Constructor

The primary purpose of the constructor is to set up some instance variables to have the proper initial values when the object is created

```
def __init__(self):
    self.x = 0
    print('I am constructed')
```

```
class PartyAnimal:

    def __init__(self):
      self.x = 0
      print('I am constructed')

    def party(self) :
      self.x = self.x + 1
      print('So far',self.x)

    def __del__(self):
      print('I am destructed', self.x)

an = PartyAnimal()
an.party()
an.party()
an = 42
print('an contains',an)
```

Running the code:

```
I am constructed
So far 1
So far 2
I am destructed 2
an contains 42
```

The constructor and destructor are optional. The constructor is typically used to set up variables. The destructor is seldomly used.

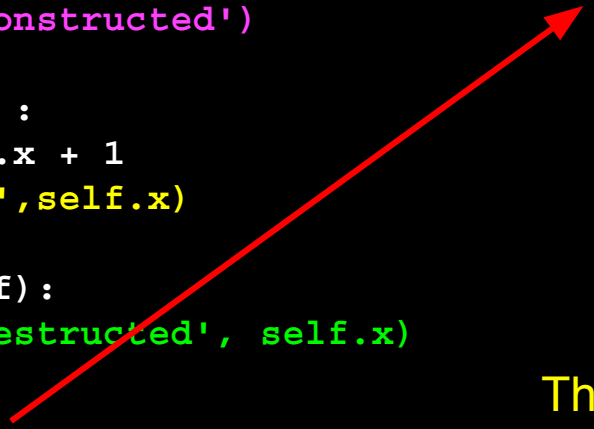
```
class PartyAnimal:

    def __init__(self):
      self.x = 0
      print('I am constructed')

    def party(self) :
      self.x = self.x + 1
      print('So far',self.x)

    def __del__(self):
      print('I am destructed', self.x)

an = PartyAnimal()
an.party()
an.party()
an = 42
print('an contains',an)
```



Running the code:

```
I am constructed
So far 1
So far 2
I am destructed 2
an contains 42
```

The constructor and destructor are optional. The constructor is typically used to set up variables. The destructor is seldom used.

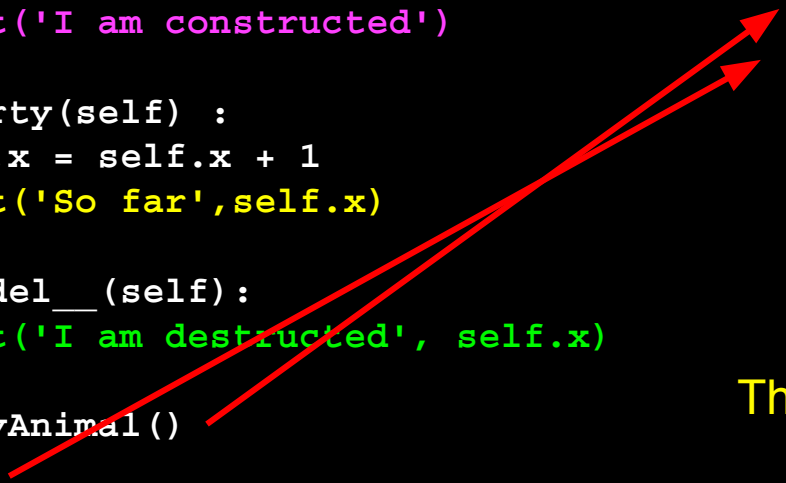
```
class PartyAnimal:

    def __init__(self):
      self.x = 0
      print('I am constructed')

    def party(self) :
      self.x = self.x + 1
      print('So far',self.x)

    def __del__(self):
      print('I am destructed', self.x)

an = PartyAnimal()
an.party()
an.party()
an = 42
print('an contains',an)
```



Running the code:

```
I am constructed
So far 1
So far 2
I am destructed 2
an contains 42
```

The constructor and destructor are optional. The constructor is typically used to set up variables. The destructor is seldom used.

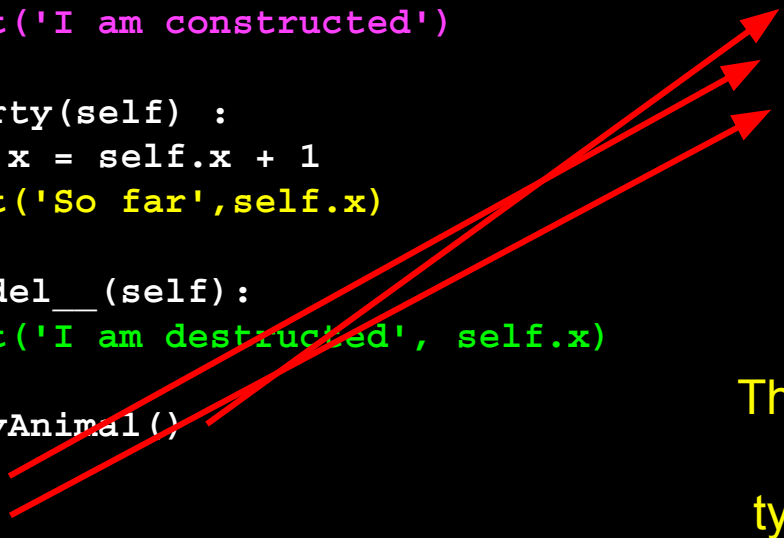
```
class PartyAnimal:

    def __init__(self):
      self.x = 0
      print('I am constructed')

    def party(self) :
      self.x = self.x + 1
      print('So far',self.x)

    def __del__(self):
      print('I am destructed', self.x)

an = PartyAnimal()
an.party()
an.party()
an = 42
print('an contains',an)
```



Running the code:

```
I am constructed
So far 1
So far 2
I am destructed 2
an contains 42
```

The constructor and destructor are optional. The constructor is typically used to set up variables. The destructor is seldom used.

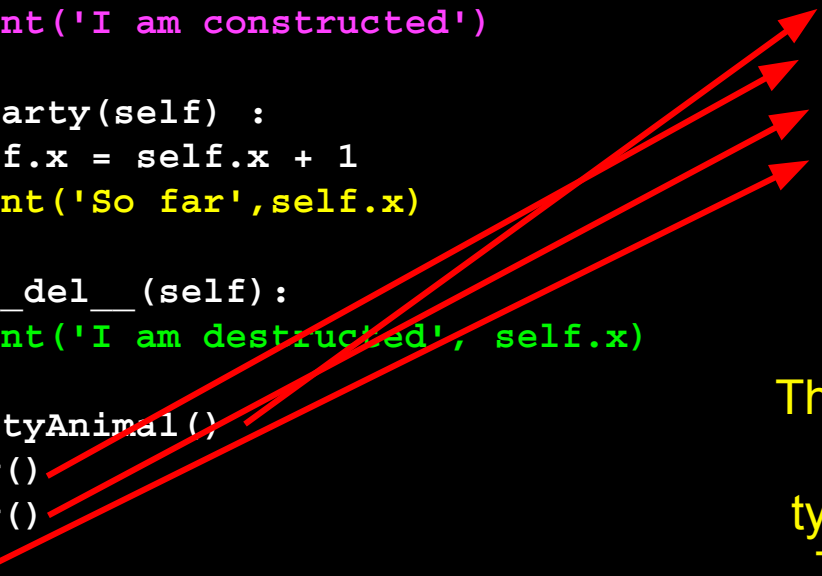
```
class PartyAnimal:

    def __init__(self):
      self.x = 0
      print('I am constructed')

    def party(self) :
      self.x = self.x + 1
      print('So far',self.x)

    def __del__(self):
      print('I am destructed', self.x)

an = PartyAnimal()
an.party()
an.party()
an = 42
print('an contains',an)
```



Running the code:

```
I am constructed
So far 1
So far 2
I am destructed 2
an contains 42
```

The constructor and destructor are optional. The constructor is typically used to set up variables. The destructor is seldom used.

```
class PartyAnimal:

    def __init__(self):
      self.x = 0
      print('I am constructed')

    def party(self) :
      self.x = self.x + 1
      print('So far',self.x)

    def __del__(self):
      print('I am destructed', self.x)

an = PartyAnimal()
an.party()
an.party()
an = 42
print('an contains',an)
```

Running the code:

```
I am constructed
So far 1
So far 2
I am destructed 2
an contains 42
```

The constructor and destructor are optional. The constructor is typically used to set up variables. The destructor is seldom used.

Many Instances

- We can create lots of objects - the class is the template for the object
- We can store each distinct object in its own variable
- We call this having multiple instances of the same class
- Each instance has its own copy of the instance variables

```
class PartyAnimal:

    def __init__(self, z):
      self.x = 0
      self.name = z
      print(self.name,"constructed")

    def party(self) :
      self.x = self.x + 1
      print(self.name,"party count",self.x)

s = PartyAnimal("Sally")
s.party()
j = PartyAnimal("Jim")

j.party()
s.party()
```

Instance 1

Instance 2

Constructors can have additional parameters. These can be used to set up instance variables for the particular instance of the class (i.e., for the particular object).

```
class PartyAnimal:

    def __init__(self, z):
      self.x = 0
      self.name = z
      print(self.name,"constructed")

    def party(self) :
      self.x = self.x + 1
      print(self.name,"party count",self.x)

s = PartyAnimal("Sally")
s.party()
j = PartyAnimal("Jim")

j.party()
s.party()
```

```
Sally constructed
Sally party count 1
Jim constructed
Jim party count 1
Sally party count 2
```

Activity: Build an Animal Class

- Live coding - let's build this together
- Step 1 - create a class called Animal
- Step 2 - give it two attributes: name and sound
- Step 3 - add a method speak() that prints "<name> says <sound>"
- Step 4 - create two Animal objects and call speak() on each

```
class Animal:

    def __init__(self, name, sound):
      self.name = name
      self.sound = sound

    def speak(self):
      print(self.name, "says", self.sound)

cat = Animal("Whiskers", "Meow")
dog = Animal("Rex", "Woof")
cat.speak()
dog.speak()
```

Animal class

- self.name and self.sound store data - these are attributes
- speak() is a method - it does something with that data
- cat and dog are two separate instances of Animal

Inheritance

Inheritance

- When we make a new class - we can reuse an existing class and inherit all the capabilities of an existing class

```
class PartyAnimal:

   def __init__(self, nam):
     self.x = 0
     self.name = nam
     print(self.name,"constructed")

   def party(self) :
     self.x = self.x + 1
     print(self.name,"party count",self.x)

class FootballFan(PartyAnimal):

   def __init__(self, nam) :
       super().__init__(nam)
       self.points = 0

   def touchdown(self):
      self.points = self.points + 7
      self.party()
      print(self.name,"points",self.points)
```

FootballFan *inherits* from PartyAnimal.
It has:

1. All the capabilities of PartyAnimal
2. Some code just for itself (the method touchdown)

```
class PartyAnimal:

   def __init__(self, nam):
     self.x = 0
     self.name = nam
     print(self.name,"constructed")

   def party(self) :
     self.x = self.x + 1
     print(self.name,"party count",self.x)

class FootballFan(PartyAnimal):

   def __init__(self, nam) :
       super().__init__(nam)
       self.points = 0

   def touchdown(self):
      self.points = self.points + 7
      self.party()
      print(self.name,"points",self.points)
```

```
s = PartyAnimal("Sally")
s.party()

j = FootballFan("Jim")
j.party()
j.touchdown()
```

Running the code above:

```
Sally constructed
Sally party count 1
Jim constructed
Jim party count 1
Jim points 7
```

Using super()

- super() - gives a child class access to its parent class
- super().__init__() - calls the parent's constructor
- Why bother - it reuses the parent's setup code instead of repeating it
- Watch out - skip super().__init__() and the parent's attributes never get set

Activity: Extend Animal to Dog

- Step 1 - create a class Dog that inherits from Animal
- Step 2 - Dog's __init__ takes name and breed, sound is always "Woof"
- Step 3 - call super().__init__() to set up name and sound
- Step 4 - add a new method fetch() that prints "<name> fetches the ball!"
- Step 5 - create a Dog object and call speak() and fetch()

Definitions

- Class - a template
- Attribute - a variable within a class
- Method - a function within a class
- Object - a particular instance of a class
- Constructor - code that runs when an object is created
- Inheritance - the ability to extend a class to make a new class.

Summary

- Object Oriented Programming is a very structured approach to code reuse
- We can group data and functionality together and create many independent instances of a class

Today:

Mon	7/13	▶ Part 1: Classes and Objects ▶ Part 2: Motivation, Player Types **Guest Lecturer: Erica Goodwin**	**Coding practice** Open work time

9 - 10: Part I - Classes and Objects

10 - 10:15: Break

10:15 - 11:30: Part II - Motivation, Player Types

(Guest Lecturer Erica Goodwin)

13 - 15: Common Mistakes, Coding Practice, open work time

Acknowledgements / Contributions

Initial Development: Charles Severance, University of Michigan School of Information

Adapted by Minh Tran, University of Chicago, Department of Computer Science and Han Dang - TA, for use in the pre-college course Creating Computer Games for Learning (Summer 2026).

...