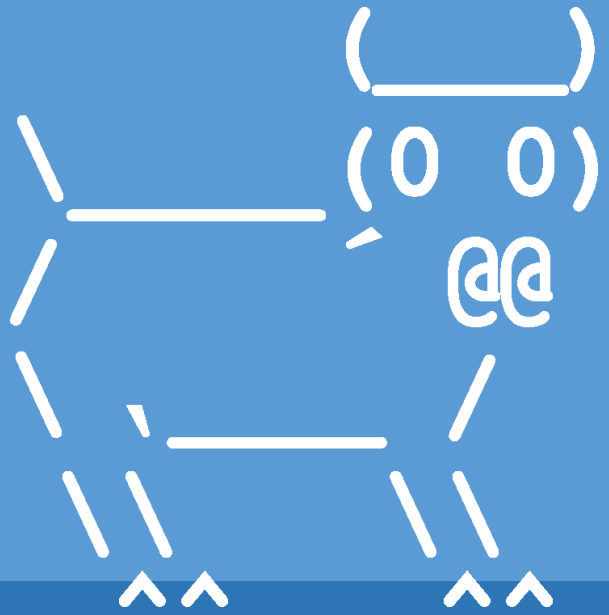
Inheritance



Introduction to Computer Science

Iris Howley

TODAY'S LESSON

Inheritance

(A hierarchy of objects for leveraging parents' implementation)

Inheritance Syntax

Super class

```
class Robot:
    __slots__ = ['name']
    def __init__(self, nm):
        self.name = nm
    def introduce(self):
        return 'I AM ' + self.name.upper()
```

Declares the super class

```
class EvilRobot(Robot):
    morality = 'evil'
```

`EvilRobot` doesn't have an initializer with a parameter!

```
>>> er1 = EvilRobot('Herbert')
>>> er1.name
'Herbert'
```

But it seems to have a name attribute

Inheritance Syntax

Super class

```
class Robot:
    __slots__ = ['name']
    def __init__(self, nm):
        self.name = nm
    def introduce(self):
        return 'I AM ' + self.name.upper()
```

Declares the super class

```
class EvilRobot(Robot):
    morality = 'evil'
```

`EvilRobot` doesn't have an initializer with a parameter!

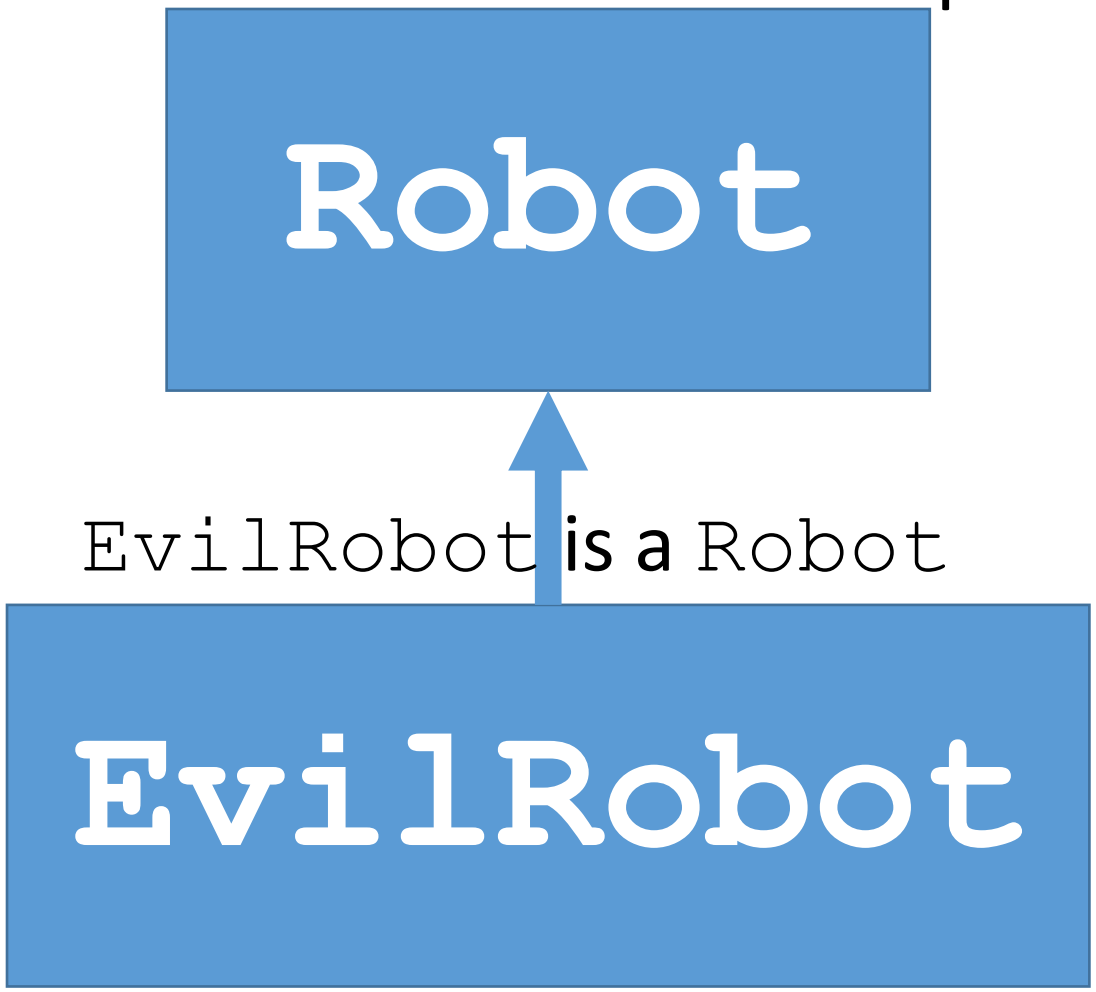
```
>>> er1 = EvilRobot('Herbert')
>>> print(er1.introduce())
I AM HERBERT
```

`EvilRobot` doesn't have an introduce() method

But `Robot` does, and it is calling that!

Class Diagram: Robot

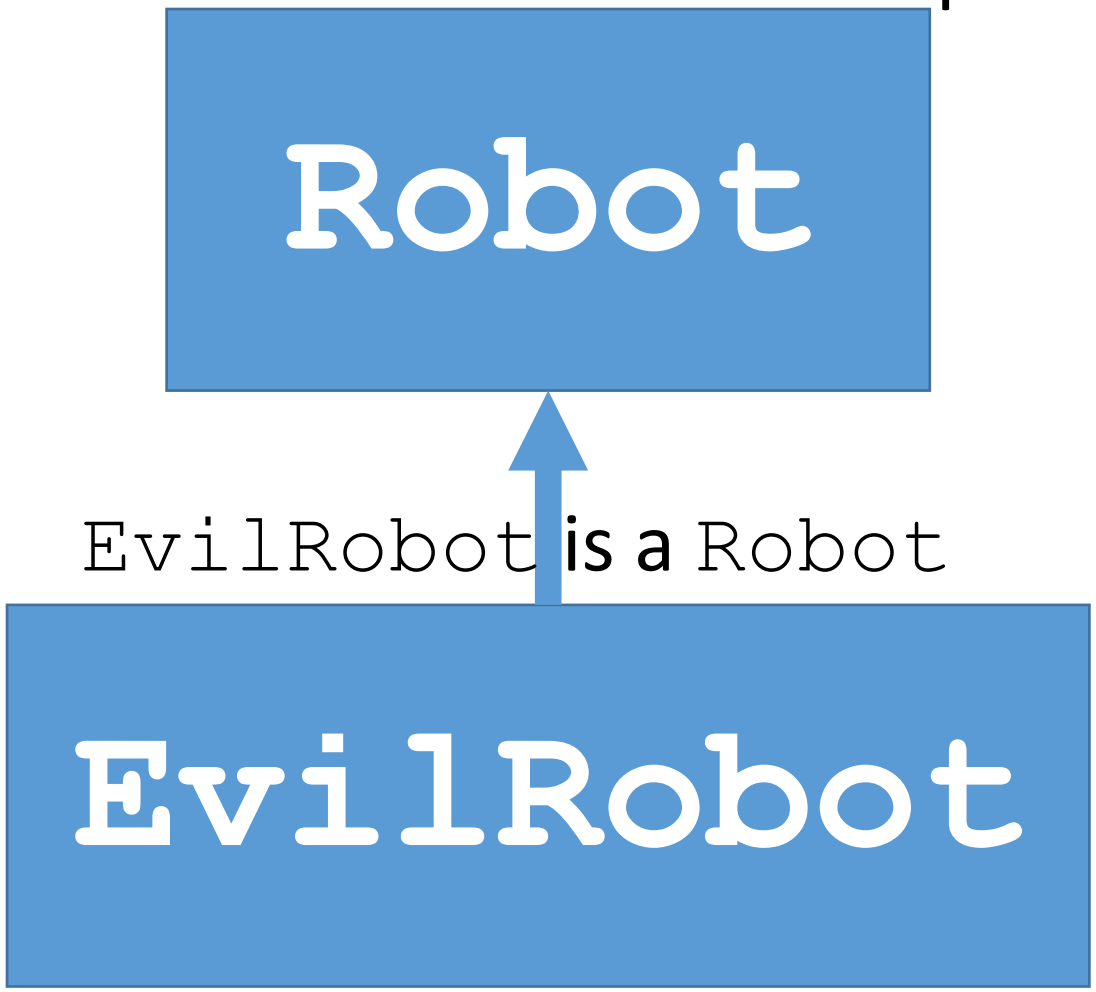
`Robot` is `EvilRobot's` super-class



`EvilRobot` is a sub-class of `Robot`

Class Diagram: Robot

`Robot` is `EvilRobot's` super-class



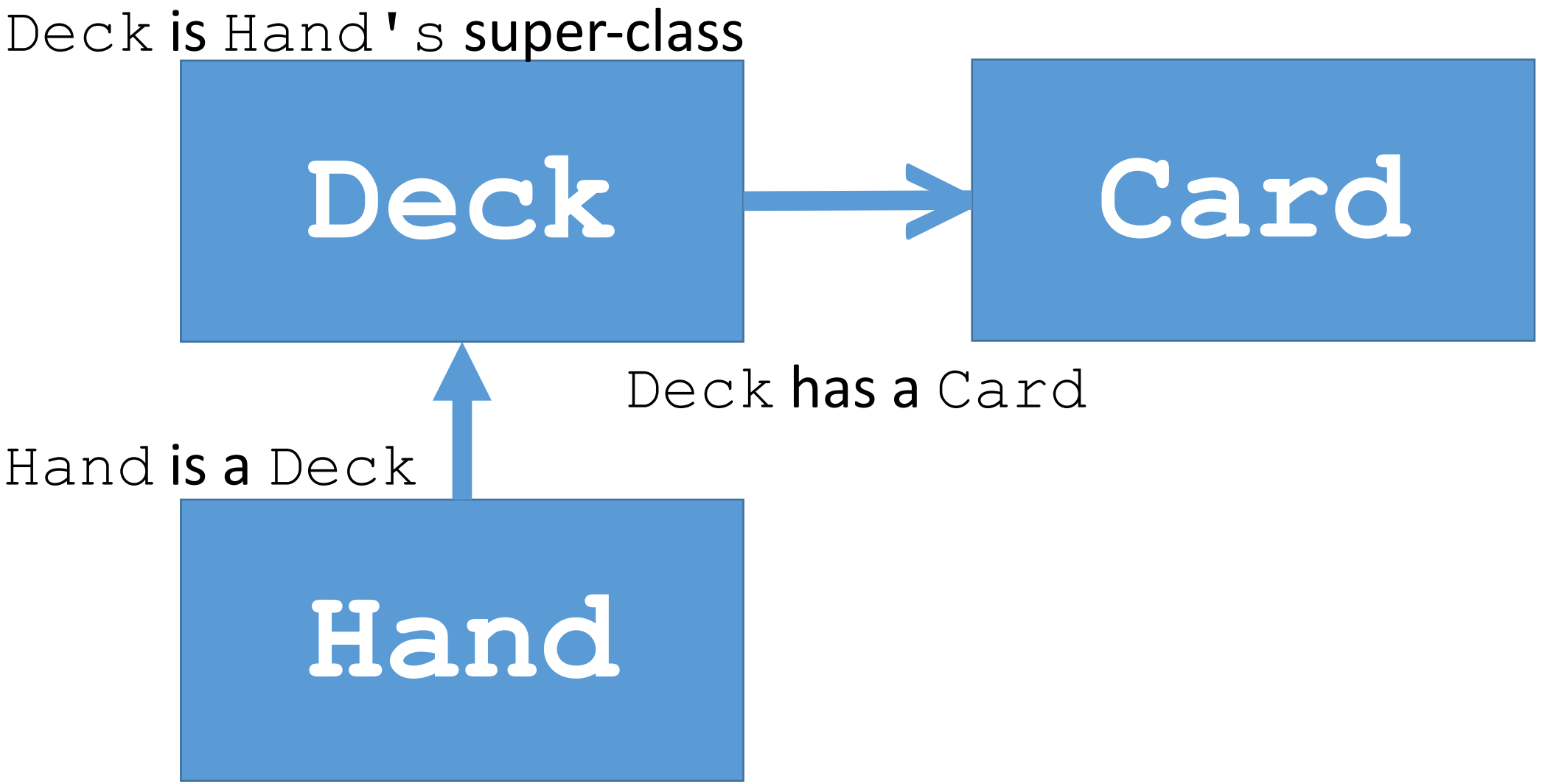
`EvilRobot` is a sub-class of `Robot`

A sub-class inherits methods, attributes from its super-class.

But a super-class does NOT inherit from its sub-classes.

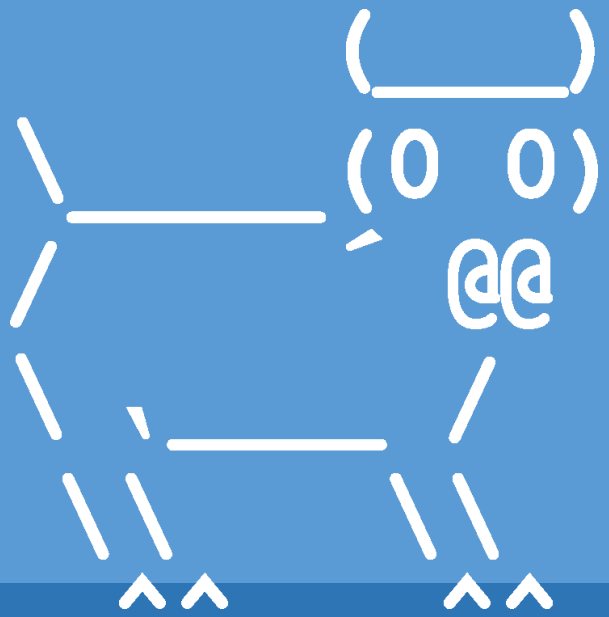
i.e., "A child inherits from the parent."

Class Diagram: Playing Cards



Please contact me!

Super Methods



Introduction to Computer Science

Iris Howley

TODAY'S LESSON

Super Methods

(Making use of the super class' methods)

```
class Robot:
    __slots__ = ['name'] # instance attribute

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

class EvilRobot(Robot):
    morality = 'evil' # class attribute
    __slots__ = ['mission']

    def __init__(self, misn):
        self.mission = misn
```

What happens when both super & sub-class have `init` method?

```
if __name__ == '__main__':
    er1 = EvilRobot('Name or Mission?!')
    print(er1.name)
AttributeError: 'EvilRobot' object has no attribute 'name'
    print(er1.mission)

    Name or Mission?!
```

The instance's `__init__` method will be called (i.e., `EvilRobot.__init(self, misn)`)

When a sub-class has the same method with same number of parameters as super-class, the sub-class' method will be called.

What if I want both the sub- and super- class methods to be called?

```
class Robot:
    __slots__ = ['name'] # instance attribute

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

class EvilRobot(Robot):
    morality = 'evil'
    __slots__ = ['mission']

    def __init__(self, nm, misn):
        super().__init__(nm)
        self.mission = misn
```

We can call the super class' __init__ method explicitly, if we want to use what's happening in it. (i.e., give all robots a name!)

```
if __name__ == '__main__':
    er1 = EvilRobot('Pearl', 'try to take over the world.')
    print(er1.name, er1.morality, er1.mission)
```

```
Pearl evil try to take over the world
```

Both `__init__` methods will be called, if we explicitly call the super class' `__init_` in the sub-class' `__init__`

We can call the super-class' methods, attributes explicitly using `super()` instead of `self`.

This is true for methods that aren't "special methods," too.

```
class Robot:
    __slots__ = ['name'] # instance attribute

    def __init__(self, nm):
        self.name = nm
    def introduce(self):
        return 'I AM ' + self.name.upper()

class EvilRobot(Robot):
    morality = 'evil' # class attribute
    __slots__ = ['mission']

    def __init__(self, nm, misn):
        Robot.__init__(self, nm)
        self.mission = misn
    def introduce(self):
        return self.mission.upper()
```

What happens when both super & sub-class have the same method (w. same # parameters)?

```
if __name__ == '__main__':
    er1 = EvilRobot('Pearl', 'try to take over the
world.')
    print(er1.name, er1.morality, er1.mission)
```

```
Pearl evil try to take over the world
```

```
    print(er1.introduce())
```

```
TRY TO TAKE OVER THE WORLD
```

The instance's method will be called (i.e., `EvilRobot.introduce()`)

```
class Robot:
    __slots__ = ['name'] # instance attribute

    def __init__(self, nm):
        self.name = nm
    def introduce(self):
        return 'I AM ' + self.name.upper()

class EvilRobot(Robot):
    morality = 'evil' # class attribute
    __slots__ = ['mission']

    def __init__(self, nm, misn):
        Robot.__init__(self, nm)
        self.mission = misn
    def introduce(self):
        return super().introduce() + '\n' + self.mission.upper()
```

…but we can call the super class' methods explicitly!

```
if __name__ == '__main__':
    er1 = EvilRobot('Pearl', 'try to take over the world.')
    print(er1.name, er1.morality, er1.mission)
```

```
Pearl evil try to take over the world
```

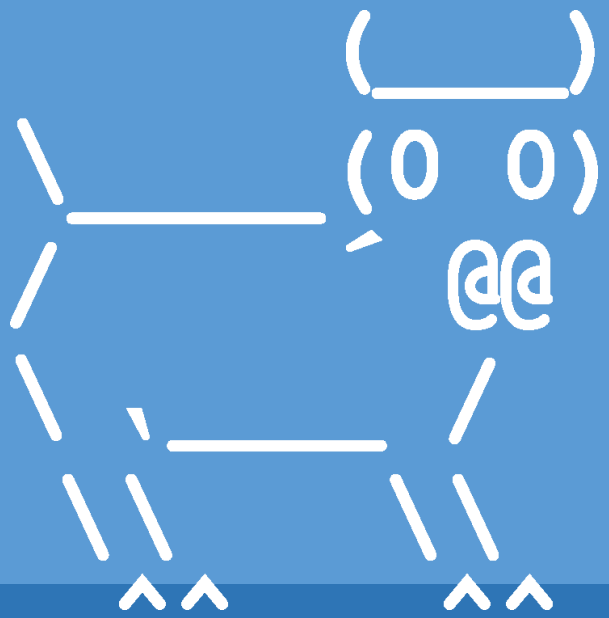
```
print(er1.introduce())
```

```
I AM PEARL
TRY TO TAKE OVER THE WORLD
```

And if we call the super class' method explicitly, it will also be called.

Please contact me!

@property.setter



Introduction to Computer Science

Iris Howley

TODAY'S LESSON

`Setter methods.`

(Like `@property`, but for modifying properties)

```
class Robot:
    __slots__ = ['_name'] # instance attribute

@property
def name(self):
        return self._name

if __name__ == '__main__'
    robit = Robot()
    robit._name = 'James Franco'
    print(robit.name)
```

@property decorator allows us to access a method like an attribute

```
class Robot:
    __slots__ = ['_name'] # instance attribute


@property
def name(self):
        return self._name


if __name__ == '__main__'
    robit = Robot()
    robit._name = 'James Franco'
    print(robit.name)
```

But to modify that attribute, we still have to access through the underscore attribute name.

UNDERSCORE GUILT!

```
class Robot:
    __slots__ = ['_name'] # instance attribute

@property
def name(self):
        return self._name
@name.setter
def name(self, value):
    self._name = value

if __name__ == '__main__'
    robit = Robot()
    robit.name = 'James Franco'
    print(robit.name)
```

We can use another decorator, `@<PROPERTY>.setter` to allow us to modify a property.

This decorator tells python which method to call when the property appears on the left-hand side of an assignment operator.

YOU CANNOT HAVE A `@property.setter` WITHOUT FIRST DEFINING THE PROPERTY WITH `@property`

WHEN WOULD WE WANT AN `@property` OR `@property.setter` TO DO SOMETHING MORE COMPLEX?

More Complex Property Methods

- Consider a temperature object
  - Has a temperature in Kelvin as an instance attribute
  - But can access the Celsius temperature equivalencies through a celsius property
  - If you change the celsius value of the Temperature object, it really just changes the Kelvin attribute appropriately

POGIL 26. Classes: Properties covers this in considerable depth.

More Complex Property Methods

```
class Temperature:
  __slots__ = ['_kelvin']
  @property
  def celsius(self):
    return self._kelvin - 273.15
  @celsius.setter
  def celsius(self, val):
    self._kelvin = val + 273.15
```

(It would make sense to make a kelvin @property, too…)

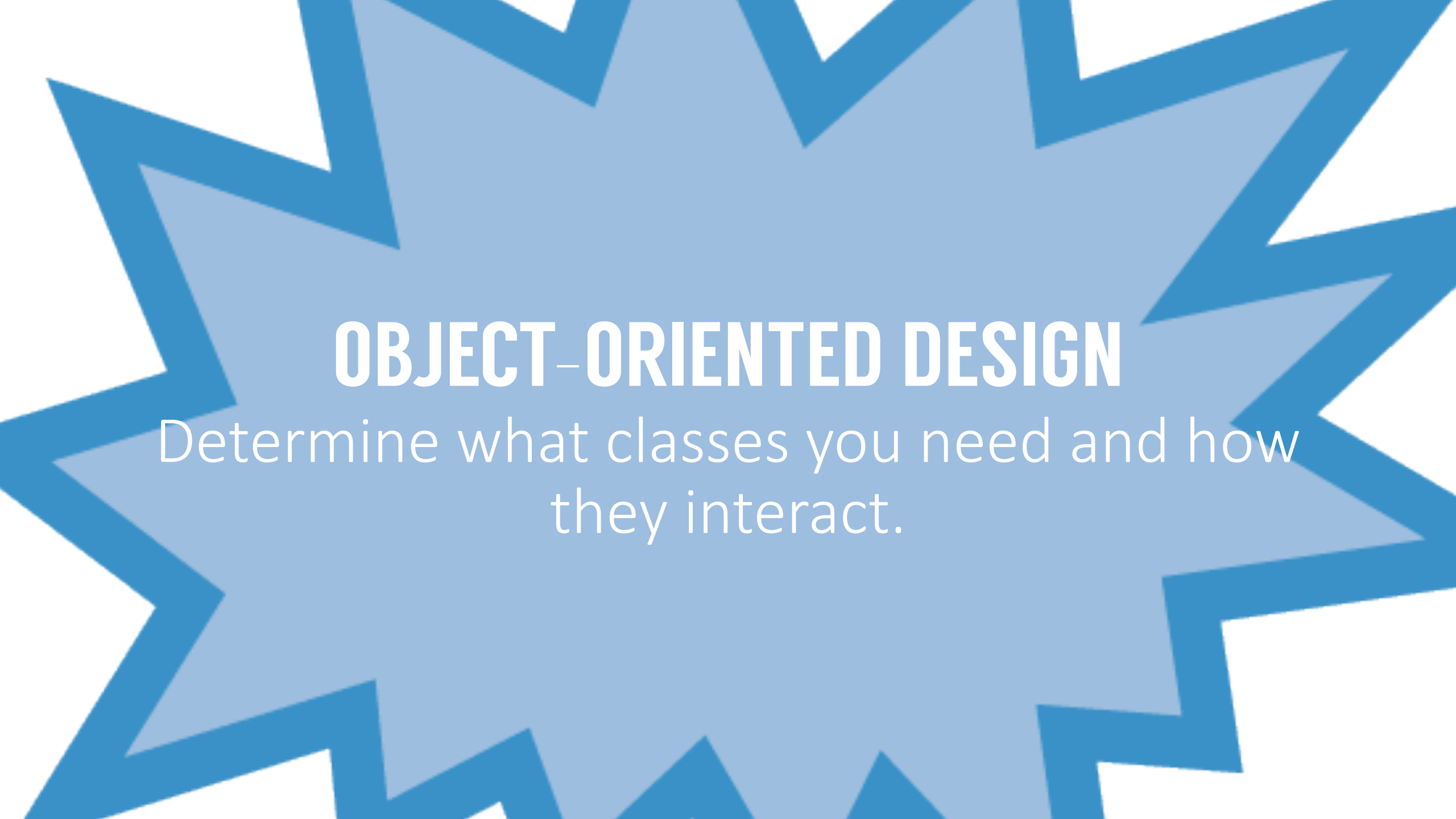
More Complex Property Methods

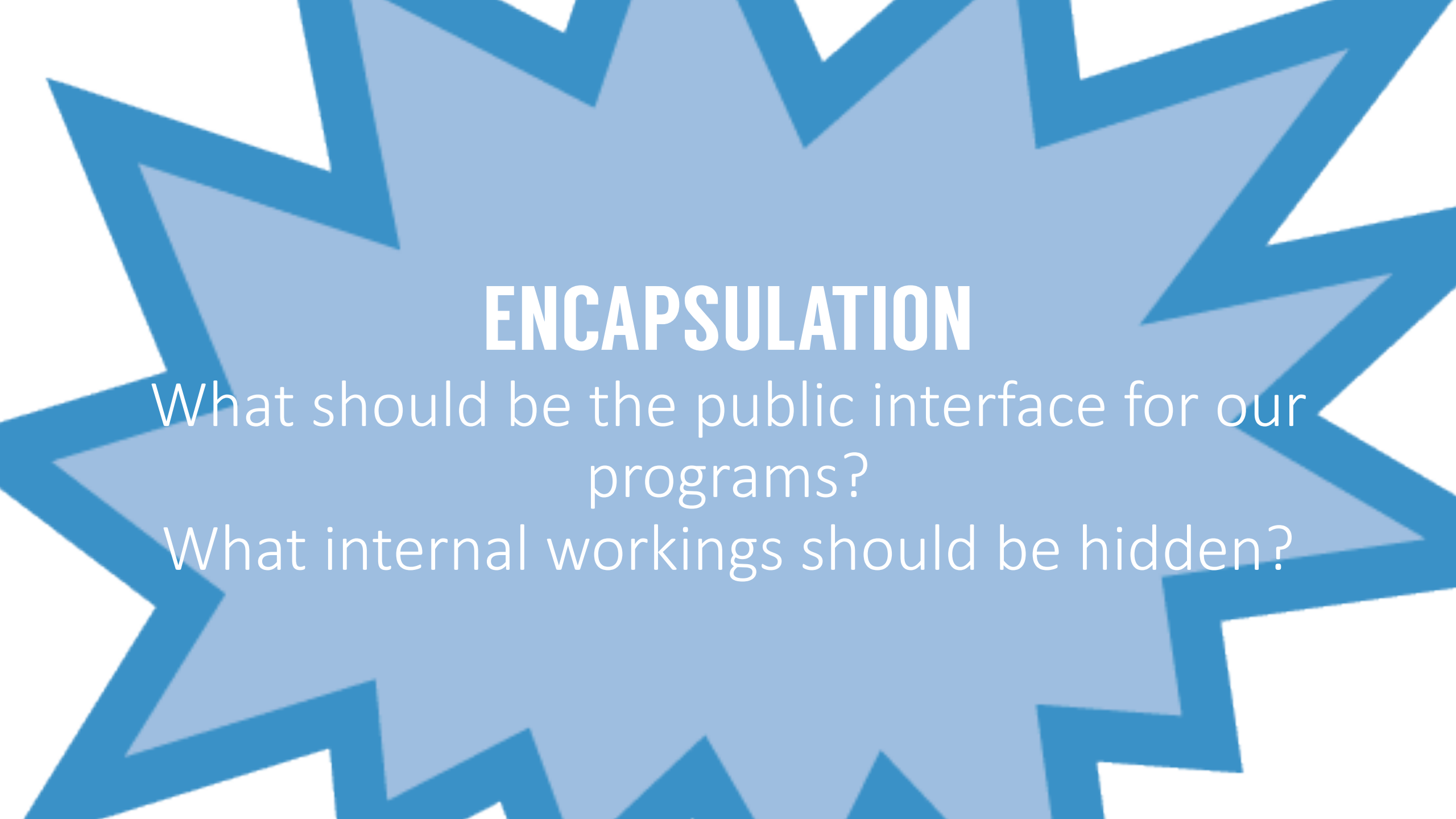
```
>>> t1 = Temperature()
>>> t1.celsius = 0  # uses @celsius.setter
>>> t1._kelvin # accesses the _kelvin attribute directly
273.15
>>> t1.celsius # uses the @property for def celsius
0.0
```

More Complex Property Methods

```
class Temperature:
  __slots__ = ['_kelvin']
  @property
  def celsius(self):
    return self._kelvin - 273.15
  @celsius.setter
  def celsius(self, val):
    self._kelvin = val + 273.15
```

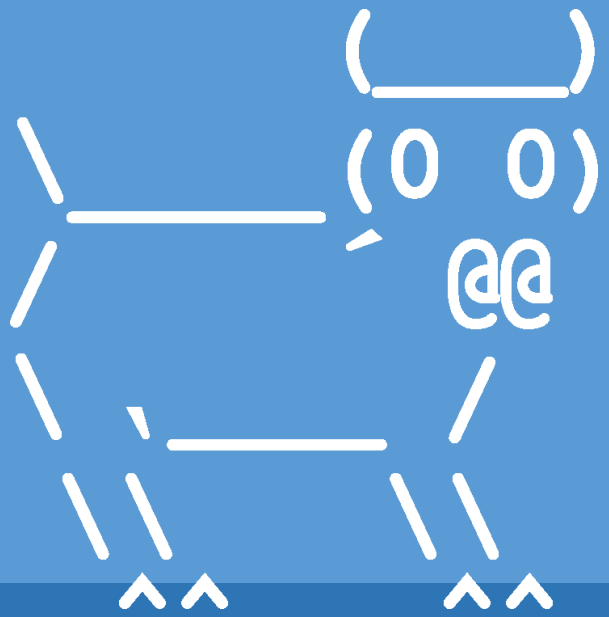
(It would make sense to make a kelvin @property, too…)


OBJECT-ORIENTED DESIGN
Determine what classes you need and how they interact.


ENCAPSULATION
What should be the public interface for our programs?
What internal workings should be hidden?

Please contact me!

Well-defined Classes



Introduction to Computer Science

Iris Howley

TODAY'S LESSON

Well-defined classes.

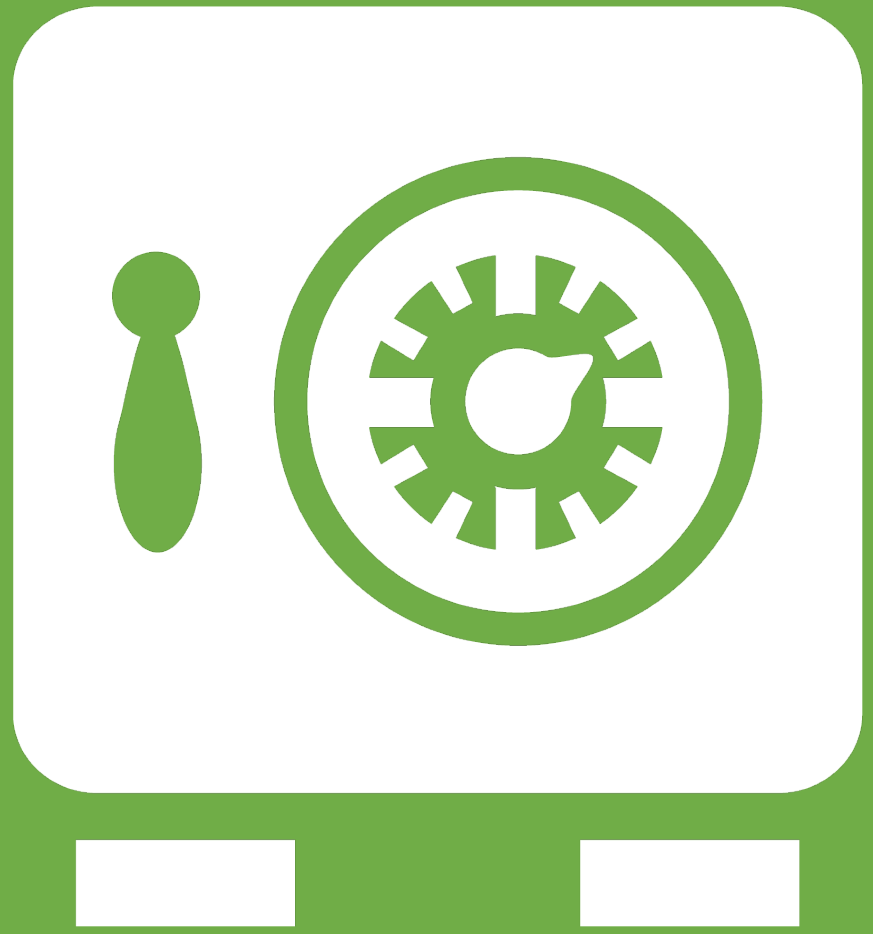
(Leveraging the class-building tools so far)

Well-defined Classes

1. Top-level docstring + every method has a docstring
   - Describe parameters and/or return values
   - In-line comments as needed
2. Meaningful variable/parameter/attribute/method names
3. `__slots__` defined to limit attributes
4. Private helper methods & attributes start with an _underscore to "hide" them
5. Private attributes that need to be accessed are given an `@property` method
6. Private attributes that need to be modified are given an `@property` method and an `@<property>.setter` method.
7. Doctests for methods
8. `__str__(self)` method, useful for debugging

See example Robot.py code on website

Please contact me!



Leftover Slides

Calling Super Methods with `super()`

```
class Robot:
    __slots__ = ['name']
    def __init__(self, nm):
        self.name = nm

class PurposeRobot(Robot):
    __slots__ = ['mission']
    def __init__(self, nm, misn):
        super().__init__(nm)
        self.mission = misn
```

`super().__init__(nm)`

is the same as:

`Robot.__init__(self, nm)`

`super()` lets us call the super-class *implicitly*…

…and we no longer need to pass `self`