Lecture 24

CLASSES PT. 2

OBJECT-ORIENTED PROGRAMMING

CSC111: Introduction to CS through Programming

R. Jordan Crouser

Assistant Professor of Computer Science

Smith College

Gradebook administrivia

- Feedback for A4 is posted
  - If you missed an assignment / lab, we applied your “free drop”
  - Extra credit points from previous assignments not yet added
  - **Result**: Moodle currently shows a conservative (but reasonably accurate) view of your standing in the course
- A few things to note:
  - Participation (10%) has not yet been calculated, and can influence your grade in **either direction**
  - Still have the final project (15%), which we will discuss in the next few weeks

Outline

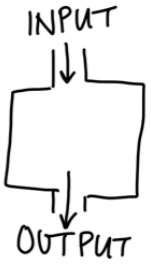
- ✓A4 recap
- ✓Friday: Classes pt. 1: attributes and methods
- Today: Classes pt. 2: object-oriented programming
  - big idea
  - recap: `classes`
  - public vs. private
- Lab: working with classes
- Wednesday: Classes pt. 3: inheritance
- Friday: Introduction to Algorithms

Remember back to the very beginning…

python™

multi-paradigm
interpreted language
with dynamic typing
and automatic memory management

object-oriented



functional

imperative

declarative

Imperative (“procedural”) programming

- Program is structured as a **set of steps** (functions and code blocks) that flow sequentially to complete a task

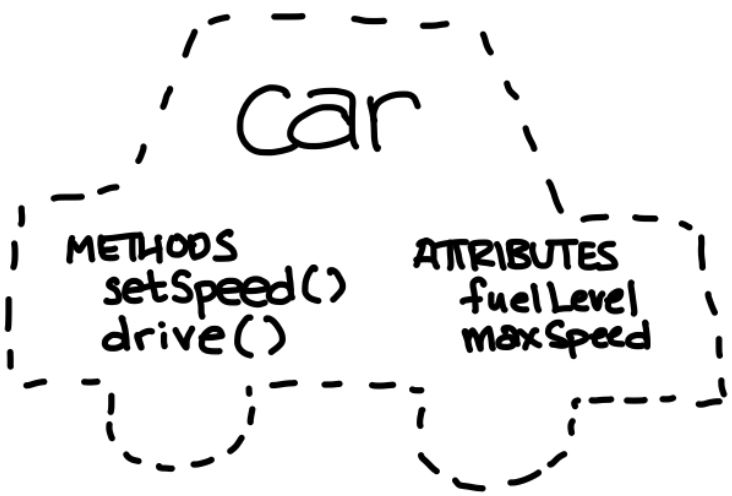
Object-oriented programming (“OOP”)

- Program is structured as a **set of objects** (with attributes and methods) that group together data and actions

Comparison: pros and cons

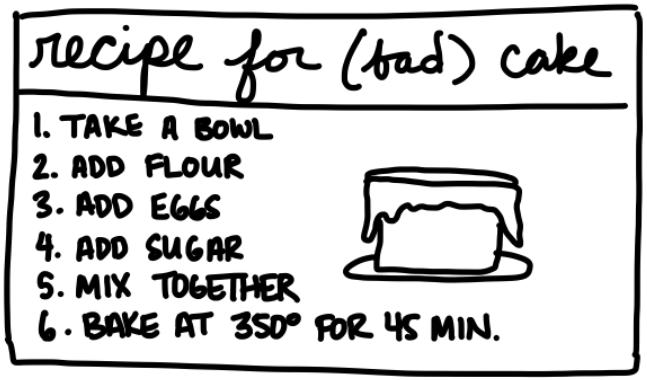
Object-oriented

(a.k.a. “OOP”)



Imperative

(a.k.a. “procedural”)



Comparison : pros and cons

	Object-oriented (a.k.a. “OOP”)	**Imperative** (a.k.a. “procedural”)
PROS	+ more organized (logically) + matches the real world + easier to test / debug + easier to reuse code	+ easy to learn and implement + only need to think a few steps ahead + much more straightforward
CONS	- more “overhead” (need to plan out further in advance) - harder to learn - overkill for small tasks	- can be hard to follow returns - have to pass stuff around - gets “unwieldy” / “clunky” - hard to test / debug

Comparison : pros and cons

	Object-oriented (a.k.a. “OOP”)	**Imperative** (a.k.a. “procedural”)
PROS	+ more organized (logically) + matches the real world + easier to test / debug + easier to reuse code	+ easy to learn and implement + only need to think a few steps ahead + much more straightforward
CONS	- more “overhead” (need to plan out further in advance) - harder to learn - overkill for small tasks	- can be hard to follow returns - have to pass stuff around - gets “unwieldy” / “clunky” - hard to test / debug

Comparison : pros and cons

	Object-oriented (a.k.a. “OOP”)	**Imperative** (a.k.a. “procedural”)
PROS	+ more organized (logically) + matches the real world + easier to test / debug + easier to reuse code	+ easy to learn and implement + only need to think a few steps ahead + much more straightforward
CONS	- more “overhead” (need to plan out further in advance) - harder to learn - overkill for small tasks	- can be hard to follow returns - have to pass stuff around - gets “unwieldy” / “clunky” - hard to test / debug

Comparison : pros and cons

	Object-oriented (a.k.a. “OOP”)	**Imperative** (a.k.a. “procedural”)
PROS	+ more organized (logically) + matches the real world + easier to test / debug + easier to reuse code	+ easy to learn and implement + only need to think a few steps ahead + much more straightforward
CONS	- more “overhead” (need to plan out further in advance) - harder to learn - overkill for small tasks	- can be hard to follow returns - have to pass stuff around - gets “unwieldy” / “clunky” - hard to test / debug

Comparison : pros and cons

	Object-oriented (a.k.a. “OOP”)	**Imperative** (a.k.a. “procedural”)
PROS	+ more organized (logically) + matches the real world + easier to test / debug + easier to reuse code	+ easy to learn and implement + only need to think a few steps ahead + much more straightforward
CONS	- more “overhead” (need to plan out further in advance) - harder to learn - overkill for small tasks	- can be hard to follow returns - have to pass stuff around - gets “unwieldy” / “clunky” - hard to test / debug

RECAP: `class` definitions ("blueprints")

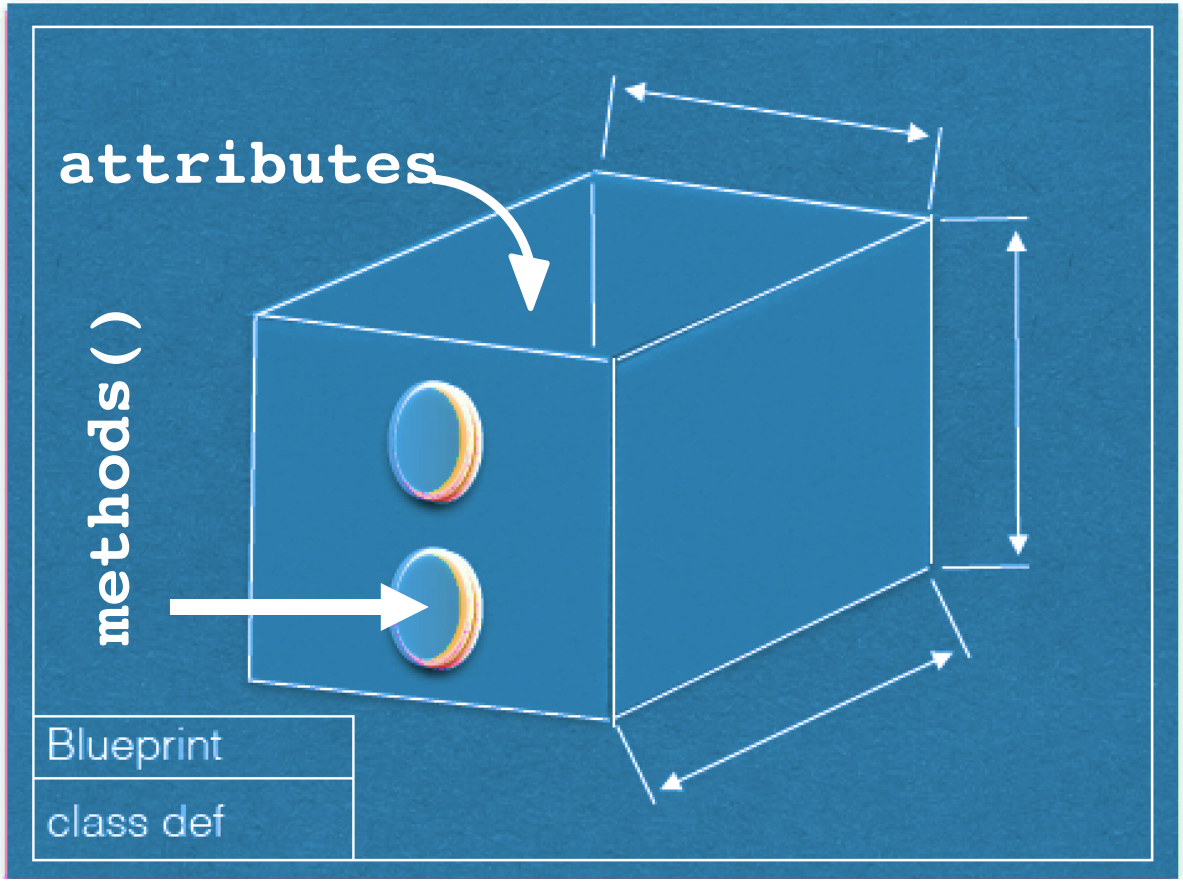
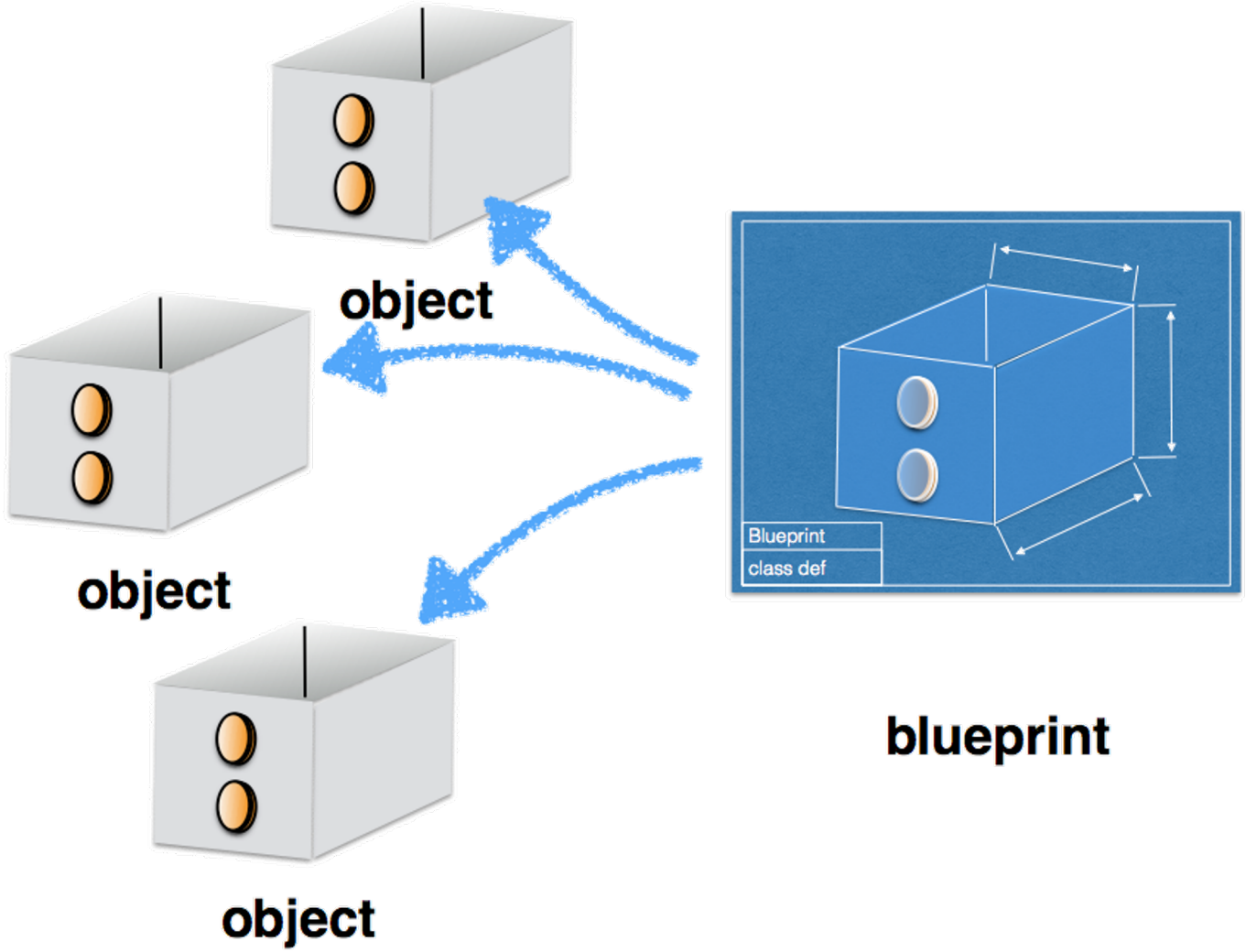


Image courtesy Dominique Thiebaut, Smith College

From a blueprint, we can make **instances**



Coding the `Die` `class`

```
from random import randint

class Die:

  def __init__(self, n_sides):
    self.num_sides = n_sides
    self.value = 1

  def roll(self):
    self.value = randint(1, self.num_sides)

  def getValue(self):
    return self.value
```

Coding the `Die` class

```
from random import randint

class Die:

    def __init__(self, n_sides):
        self.num_sides = n_sides
        self.value = 1

    def roll(self):
        self.value = randint(1, self.num_sides)

    def getValue(self):
        return self.value
```

Coding the `Die` `class`

```
from random import randint

class Die:

  def __init__(self, n_sides):
    self.num_sides = n_sides
    self.value = 1

  def roll(self):
    self.value = randint(1, self.num_sides)

  def getValue(self):
    return self.value
```

attributes

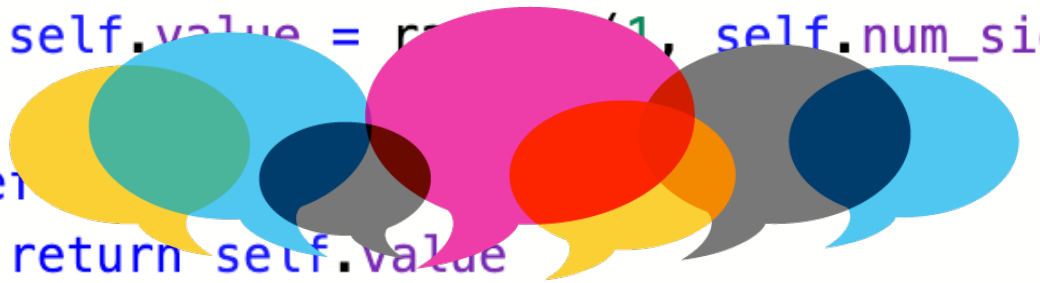
Coding the `Die` class

```
from random import randint

class Die:

  def __init__(self, n_sides):
    self.num_sides = n_sides
    self.value = 1

  def roll(self):
    self.value = randint(1, self.num_sides)

  def getValue(self):
    return self.value
```

methods

What happens if I **run** this program?

```
from random import randint

class Die:

  def __init__(self, n_sides):
    self.num_sides = n_sides
    self.value = 1

  def roll(self):
    self.value = r          (1, self.num_sides)

  def
    return self.value
```

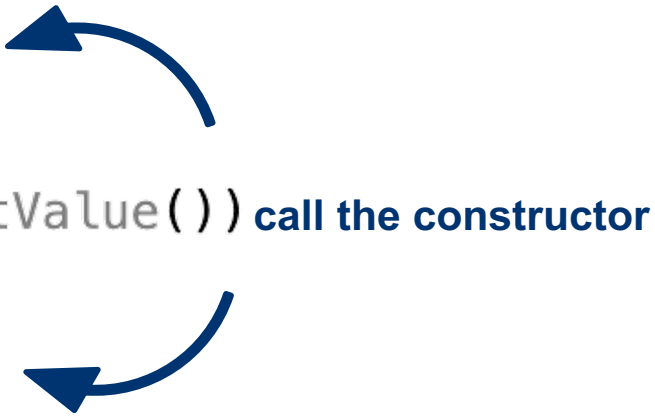


Using the class

```
def main():
    d6 = Die(6)
    d6.roll()
    print(d6.getValue())

    d8 = Die(8)
    d8.roll()
    print(d8.getValue())

if __name__ == "__main__":
    main()
```

Creating `Die` instances

```
def main():
    d6 = Die(6)
    d6.roll()
    print(d6.getValue())

    d8 = Die(8)
    d8.roll()
    print(d8.getValue())

if __name__ == "__main__":
    main()
```



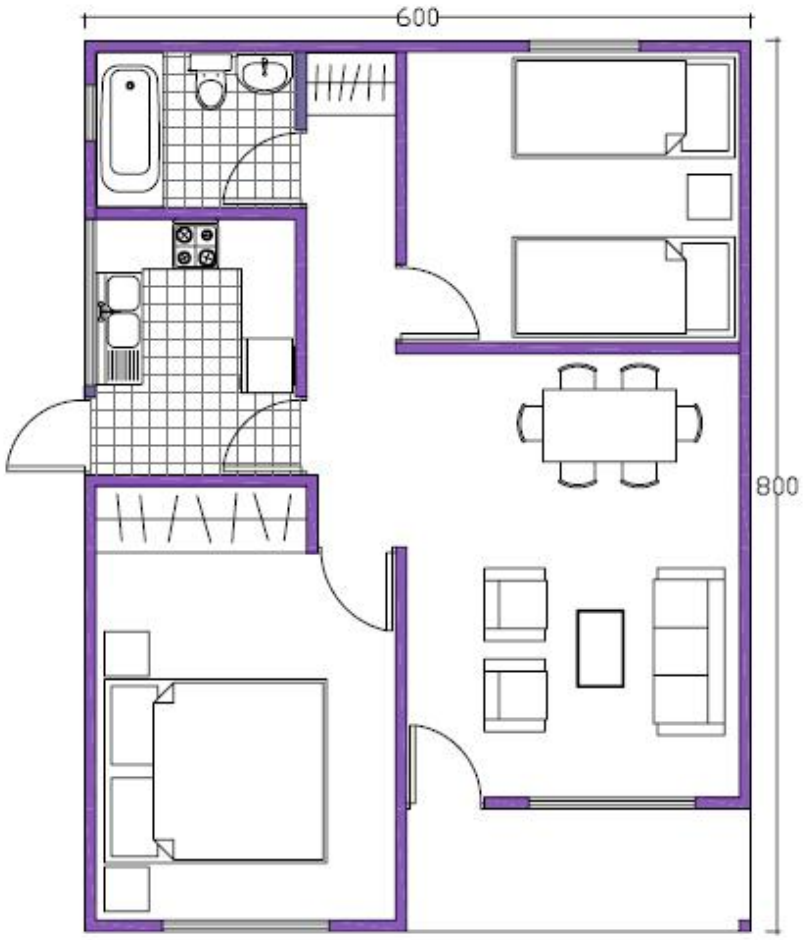
Lots of possible `Die` instances

they don't all have the same **attributes**

All from the **same blueprint**

```
from random import randint

class Die:

  def __init__(self, n_sides):
    self.num_sides = n_sides
    self.value = 1

  def roll(self):
    self.value = randint(1, self.num_sides)

  def getValue(self):
    return self.value
```

`class` definition vs. **`instance`**

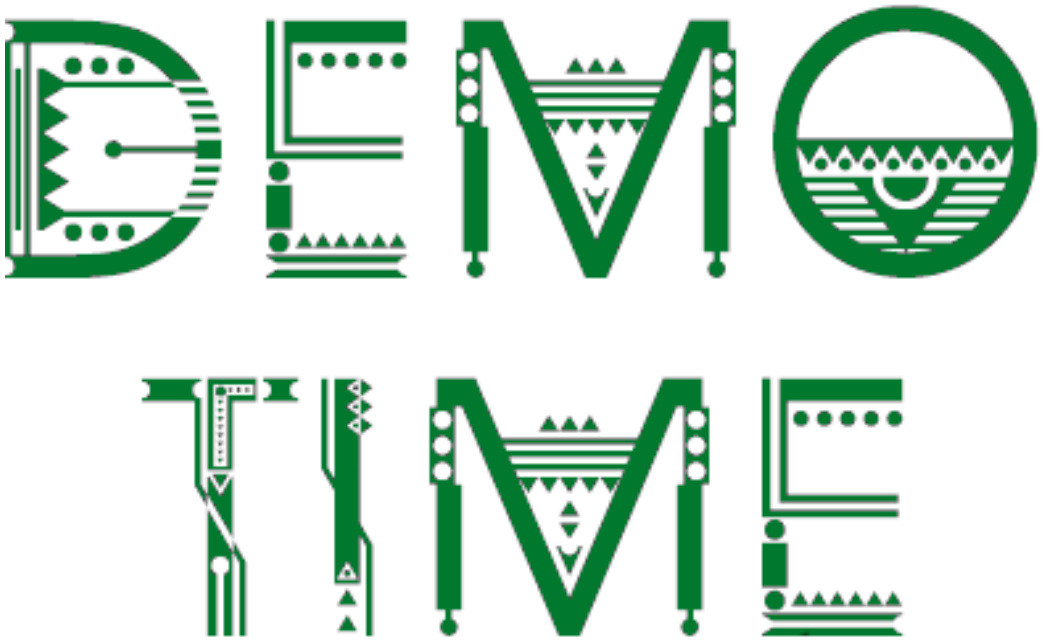


…make sense?

Lingering question

```
def getValue(self):
    return self.value
```

"Why can't I just access attributes **directly**?"


DEMO
TIME

Think back to our ATM example

Can you imagine any **`attributes/methods`** you might want to be **`private`**?

public VS. **private**

- **python methods/attributes** are **public** by default
  this means that they can be accessed from **outside** the **instance**… by anyone
  (for better or for worse)

- To make a **method/attribute** **private** (i.e. accessible only within the **instance** itself), prefix it with a double underscore (__)

```
def __init__(self, pin):
    self.__pin = pin
```

Big takeaways

- Object-oriented programming is a **powerful paradigm**
- **It's also very common** (and therefore useful to learn)
- The more **complex** your problem, the more it makes sense to **organize your code this way**
- In Python, it isn't all or nothing: some parts of your program might be object-oriented, others might be procedural
- The important part is that your code **makes sense**

Up next

- ✓A4 recap
- ✓Friday: Classes pt. 1: attributes and methods
- ✓Today: Classes pt. 2: object-oriented programming
  - ✓big idea
  - ✓recap: `classes`
  - ✓public vs. private
- Lab: working with classes
- Wednesday: Classes pt. 3: inheritance
- Friday: Introduction to Algorithms