Intro to Coding with Python– Classes Pt 2

Dr. Ab Mosca (they/them)

Slides based off slides courtesy of Jordan Crouser (https://jcrouser.github.io/)

Plan for Today

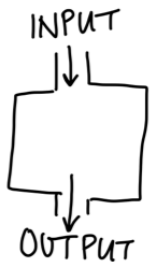
- Object-Oriented Programming
  - Big idea
  - recap classes
  - public vs private

Remember back to the very beginning…

python™

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

object-oriented



functional

imperative

declarative

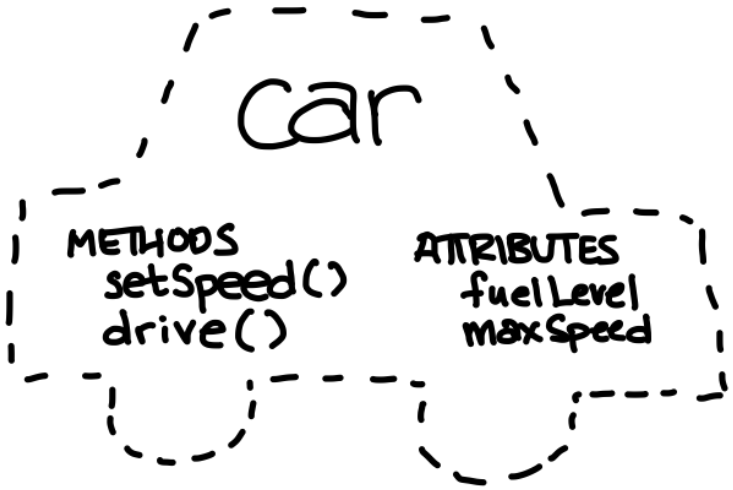
Imperative ("procedural") programming

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

Object-oriented programming ("OOP")

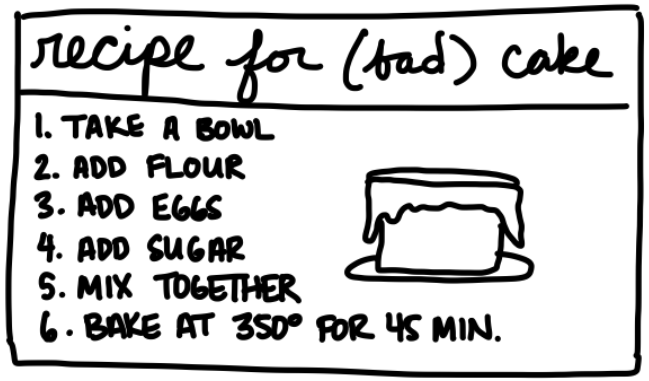
- Program is structured as a **set of objects** (with attributes and methods) that group together data and actions

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

RECAP: `class` definitions (“blueprints”)

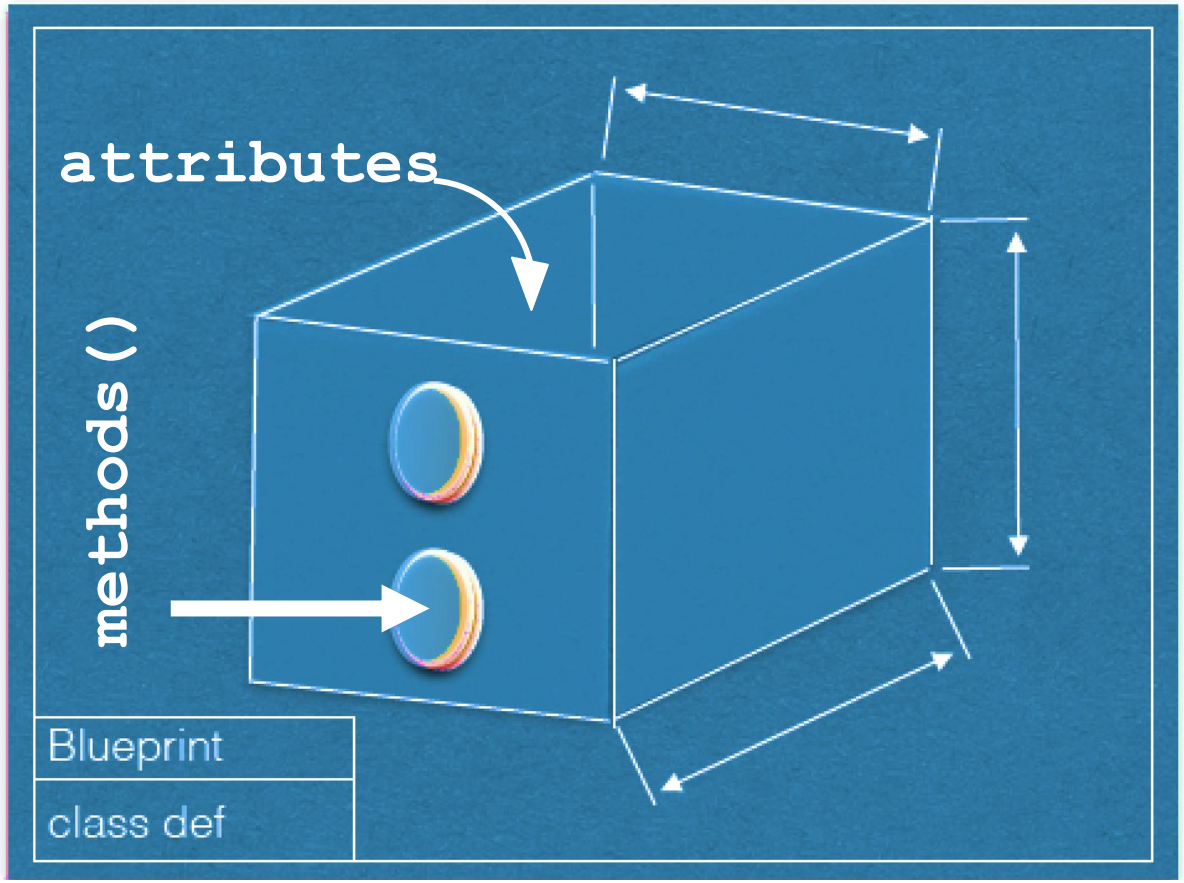
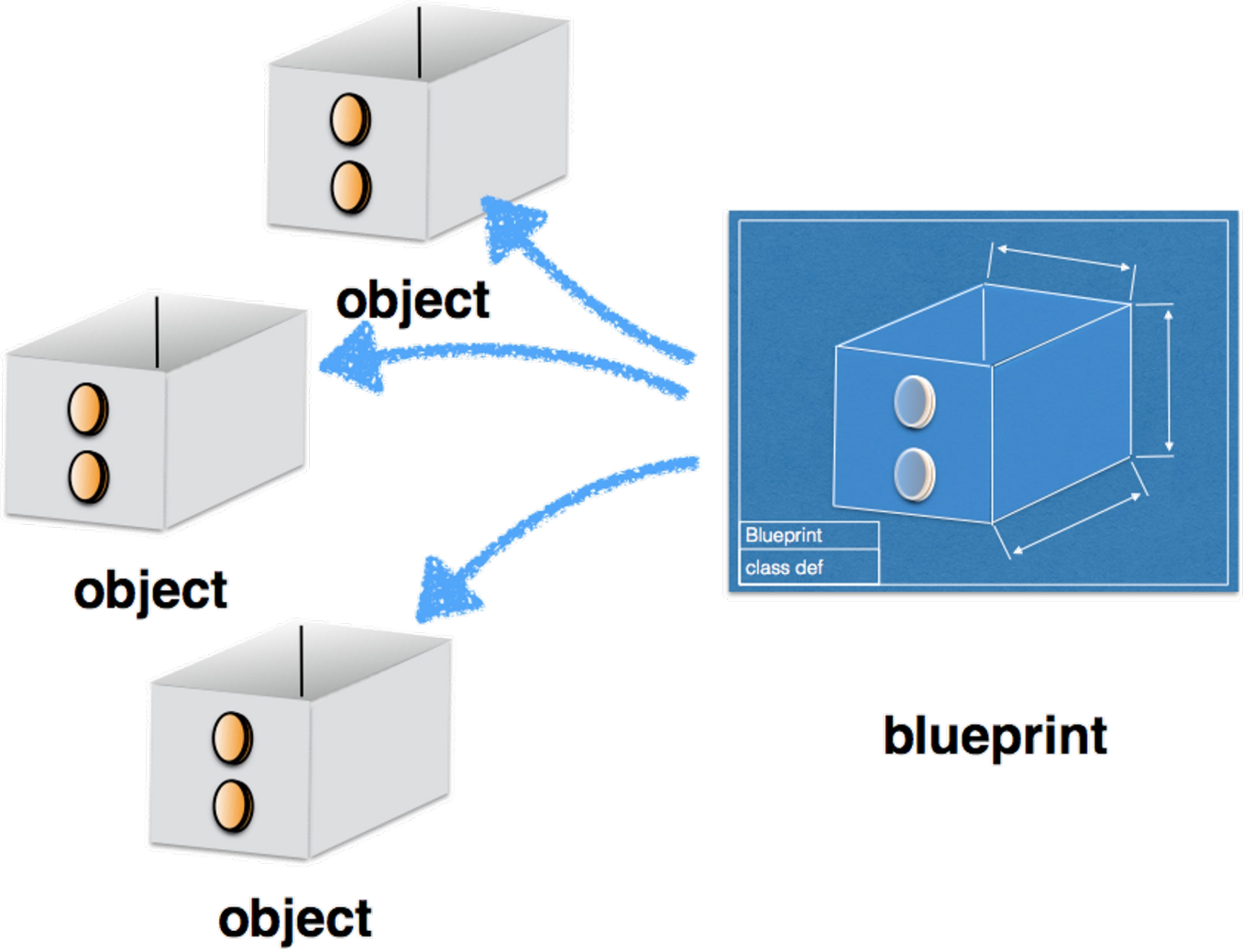


Image courtesy Dominique Thiebaut, Smith College

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



Coding the `Die` class

```python
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

the constructor

```
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?

```python
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
```

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()
```

call the constructor

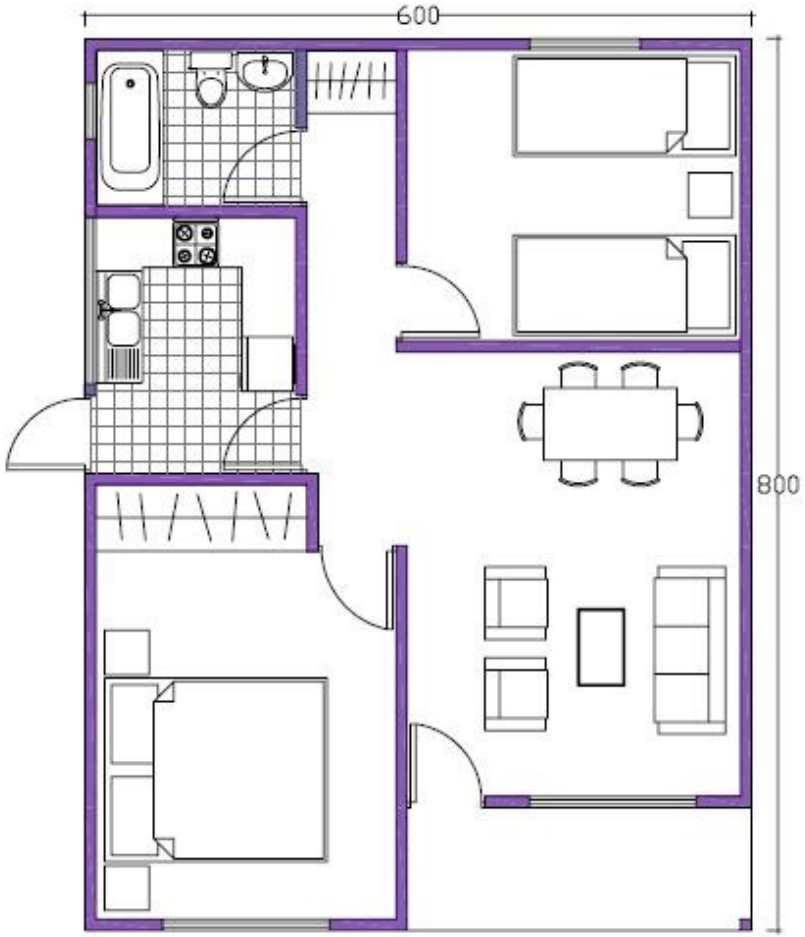
Lots of possible `Die` `instances`

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

All from the **same blueprint**

```python
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**?"

Think back to our ATM example

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

```
print(account.pin)
```

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
```

15-minute exercise

- Create a class to represent this class (CAIS 117)
- Which attributes should be private and which should be public?
- Once you have your class, write a program that makes an instance of that class and prints that names of everyone in class

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**