

Computational Structures in Data Science

Object-Oriented Programming

Berkeley
UNIVERSITY OF CALIFORNIA

Announcements

- Midterm Thurs 3/14
- We will be sending seating assignments out early next week
- If you have a conflict email us ASAP – should have already filled out the form
- cs88@berkeley.edu (Please don't email just me. I love to help, but can't get to 500 emails)
- Review sessions posted on Ed.
- Will be updating Lab/HW schedule to give you a chance to study.

Computational Structures in Data Science

Object-Oriented Programming

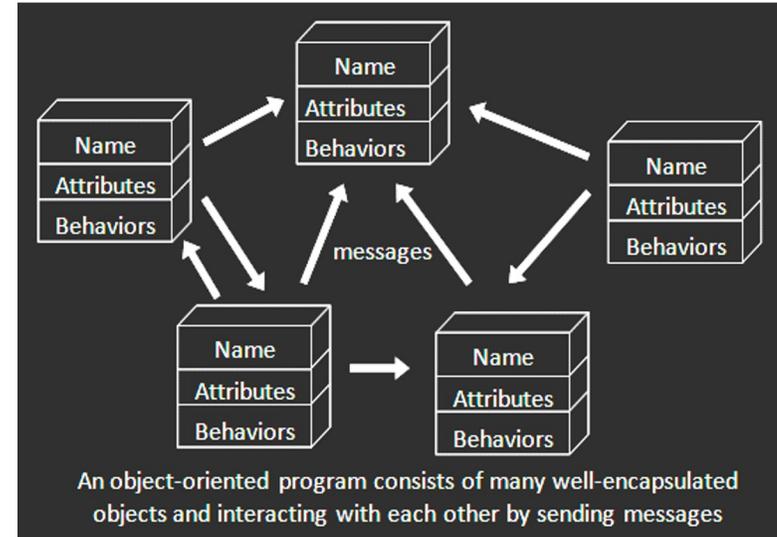
Berkeley
UNIVERSITY OF CALIFORNIA

Learning Objectives

- Learn how to make a class in Python
 - `class` keyword
 - `__init__` method
 - `self`

Object-Oriented Programming (OOP)

- **Objects** as data structures
 - With methods you ask of them
 - These are the behaviors
 - With local state, to remember
 - These are the attributes
- **Classes & Instances**
 - Instance an example of class
 - E.g., Fluffy is instance of Dog
- **Inheritance** saves code
 - Hierarchical classes
 - e.g., a Tesla is a special case of an Electric Vehicle, which is a special case of a car
- Other Examples (though not pure)
 - Java (CS61B), C++



www3.ntu.edu.sg/home/ehchua/programming/java/images/OOP-Objects.gif

Object-Oriented Programming is *About Design*

"In my version of computational thinking, I imagine an abstract machine with just the data types and operations that I want. If this machine existed, then I could write the program I want.

But it doesn't. Instead I have introduced a bunch of subproblems — the data types and operations — and I need to figure out how to implement them. I do this over and over until I'm working with a real machine or a real programming language. That's the art of design."

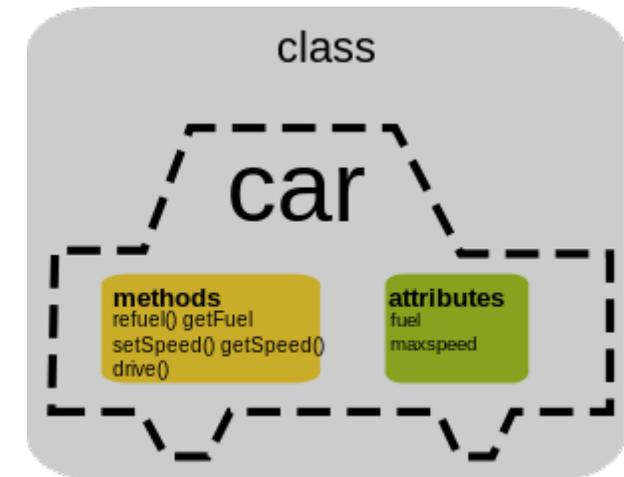
— Barbara Liskov,
Turing Award Winner, UC Berkeley '61.

[Full interview](#)



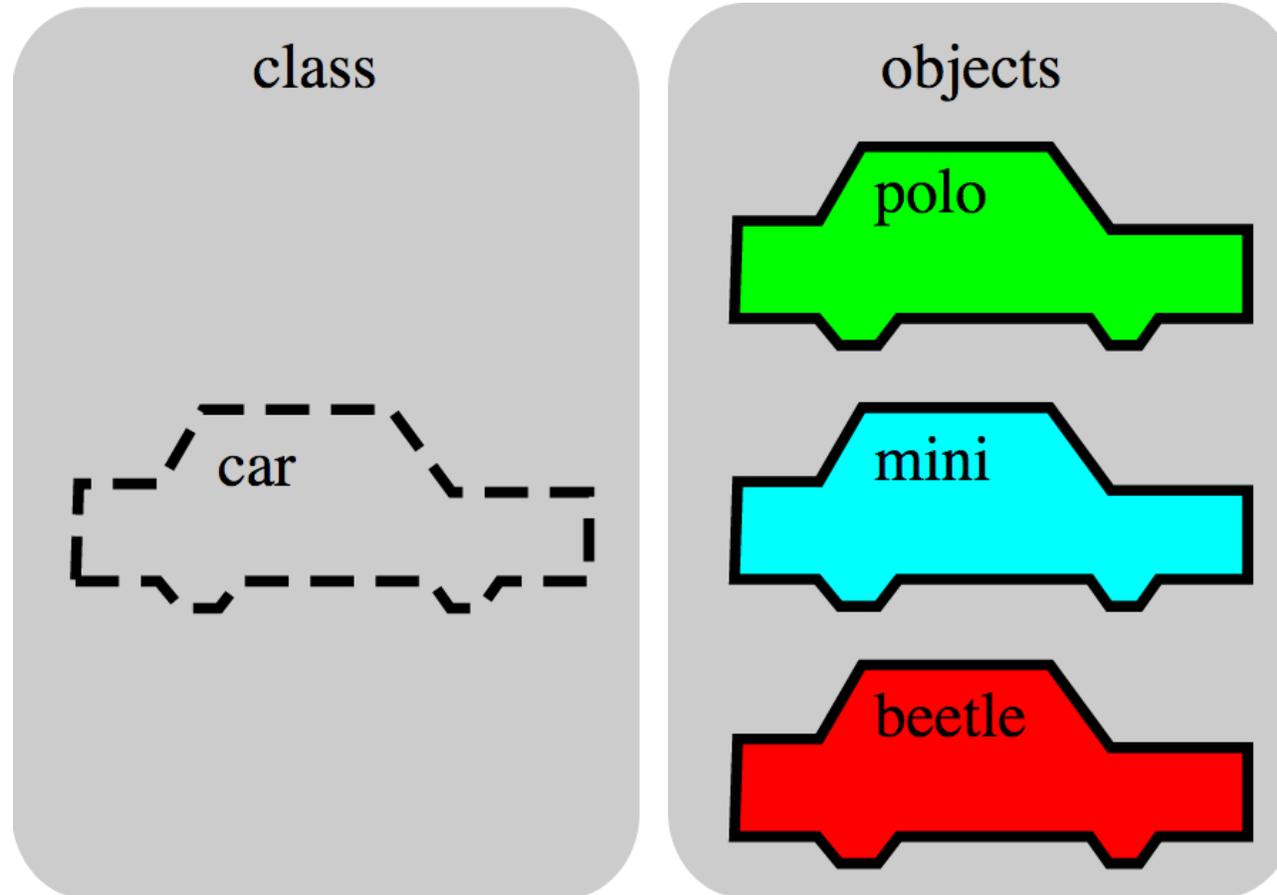
Classes

- Consist of data and behavior, bundled together to create abstractions
 - Abstract Data Types use functions to create abstractions
 - Classes define a new **type** in a programming language
 - They make the "abstract" data type concrete.
- A class has
 - attributes (variables)
 - methods (functions)
that define its behavior.



Objects

- An **object** is the instance of a class.



Objects

- Objects are concrete instances of classes in memory.
- They have *state*
 - mutable vs immutable (lists vs tuples)
- Methods are functions that belong to an object
 - Objects do a collection of **related** things
- In Python, *everything* is an object
 - All **objects** have **attributes**
 - Manipulation happens through **methods**

Python class statement

```
class ClassName:  
    def __init__(self):  
        <initialization steps>  
    .  
    .  
    .  
    <statement-N>
```

```
# Coming Next Week:  
class ClassName ( inherits ):  
    <statement-1>  
    .  
    .  
    .  
    <statement-N>
```

From ADTs to Classes

- An ADT is an *abstract* representation of a *type* of Data.

```
def points(x, y) # our point ADT
    return { 'x': x, 'y': y }
```

```
class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y
    def subtract(self, other):
        return Point(self.x - other.x, self.y -
other.y)
```

From ADTs to Classes (Usage)

```
>>> origin = point(0, 0)    # Using the ADT
>>> type(origin)
<class 'dict'>
>>> origin
{'x': 0, 'y': 0}
>>> my_house = Point(5, 5)  # Using the class
>>> my_house.x
5
>>> type(my_house)
<class '__main__.Point'>
>>> my_house
<__main__.Point object at 0x104fdc710>
```

Example: Account

```
class BaseAccount:
```

```
    def __init__(self, name, initial_deposit):  
        self.name = name  
        self.balance = initial_deposit  
  
    def account_name(self):  
        return self.name  
  
    def balance(self):  
        return self.balance  
  
    def withdraw(self, amount):  
        self.balance -= amount  
        return self.balance
```

new namespace

attributes

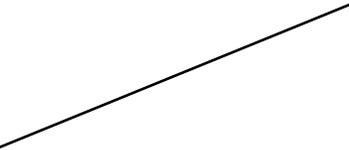
The object

dot

methods

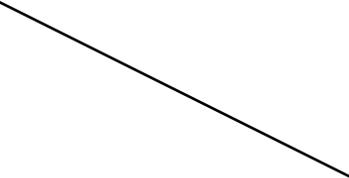
Creating an object, invoking a method

The Class Constructor



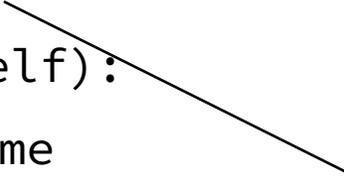
```
my_acct = BaseAccount("John Doe", 93)  
my_acct.withdraw(42)
```

dot



Special Initialization Method

```
class BaseAccount:  
  
    def __init__(self, name, initial_deposit):  
        self.name = name  
        self.balance = initial_deposit  
  
    def account_name(self):  
        return self.name  
  
    def balance(self):  
        return self.balance  
  
    def withdraw(self, amount):  
        self.balance -= amount  
        return self.balance
```



return None

More on Attributes

- Attributes of an object accessible with 'dot' notation
`obj.attr`
- You can distinguish between "public" and "private" data.
 - Used to clarify to programmers how you class should be used.
 - In Python an `_` prefix means "this thing is private"
 - `_foo` and `__foo` do different things inside a class.
 - [More for the curious.](#)
- Class variables vs Instance variables:
 - Class variable set for all instances at once
 - Instance variables per instance value

Example

```
class BaseAccount:

    def __init__(self, name, initial_deposit):
        self.name = name
        self.balance = initial_deposit

    def name(self):
        return self.name

    def balance(self):
        return self.balance

    def withdraw(self, amount):
        self.balance -= amount
        return self.balance
```

Example: Suggested “private” attributes

```
class BaseAccount:

    def __init__(self, name, initial_deposit):
        self._name = name
        self._balance = initial_deposit

    def name(self):
        return self._name

    def balance(self):
        return self._balance

    def withdraw(self, amount):
        self._balance -= amount
        return self._balance
```

Example: class attribute

```
class BaseAccount:
    account_number_seed = 1000

    def __init__(self, name, initial_deposit):
        self._name = name
        self._balance = initial_deposit
        self._acct_no = BaseAccount.account_number_seed
        BaseAccount.account_number_seed += 1

    def name(self):
        return self._name

    def balance(self):
        return self._balance

    def withdraw(self, amount):
        self._balance -= amount
        return self._balance
```

More class attributes

```
class BaseAccount:
    account_number_seed = 1000
    accounts = []

    def __init__(self, name, initial_deposit):
        self._name = name
        self._balance = initial_deposit
        self._acct_no = BaseAccount.account_number_seed
        BaseAccount.account_number_seed += 1
        BaseAccount.accounts.append(self)

    def name(self):
        ...

    def show_accounts():
        for account in BaseAccount.accounts:
            print(account.name(),
                  account.account_no(), account.balance())
```