

# Computational Structures in Data Science

---

## Object-Oriented Programming

UC Berkeley

# Announcements

- **Midterm Grades: By the end of the week**
  - Working through them as fast as possible. 😊
- **Reminder: Maps project due tonight**
- **Ants Project**
  - Out later this week!
  - **Not** expected to work over Spring break, but you may wish to get started.
  - **Find a partner!**
  - **New Requirement!** Will to "Oral Check Offs"
    - Details TBD – will do a ~5 minute check in with a TA during the final week of the project.

# Computational Structures in Data Science

---

## Object-Oriented Programming

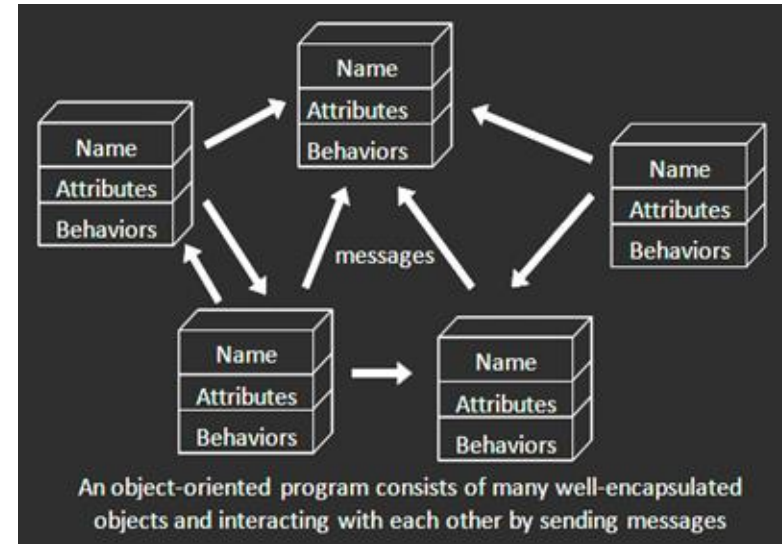
UC Berkeley

# 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](http://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."

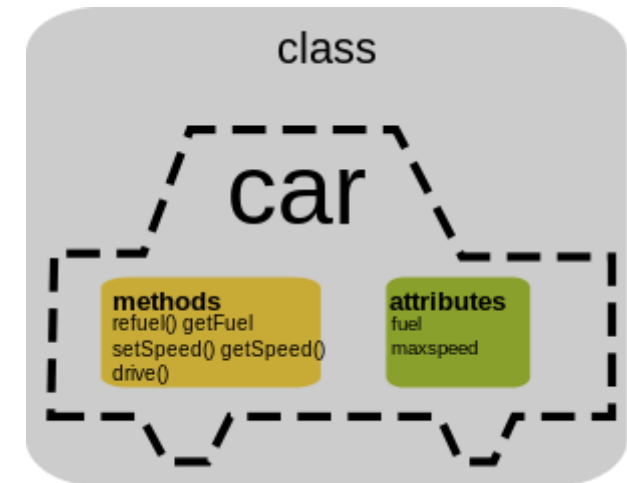
— Barbara Liskov,  
Turing Award Winner, UC Berkeley '61.

[Full interview](#)



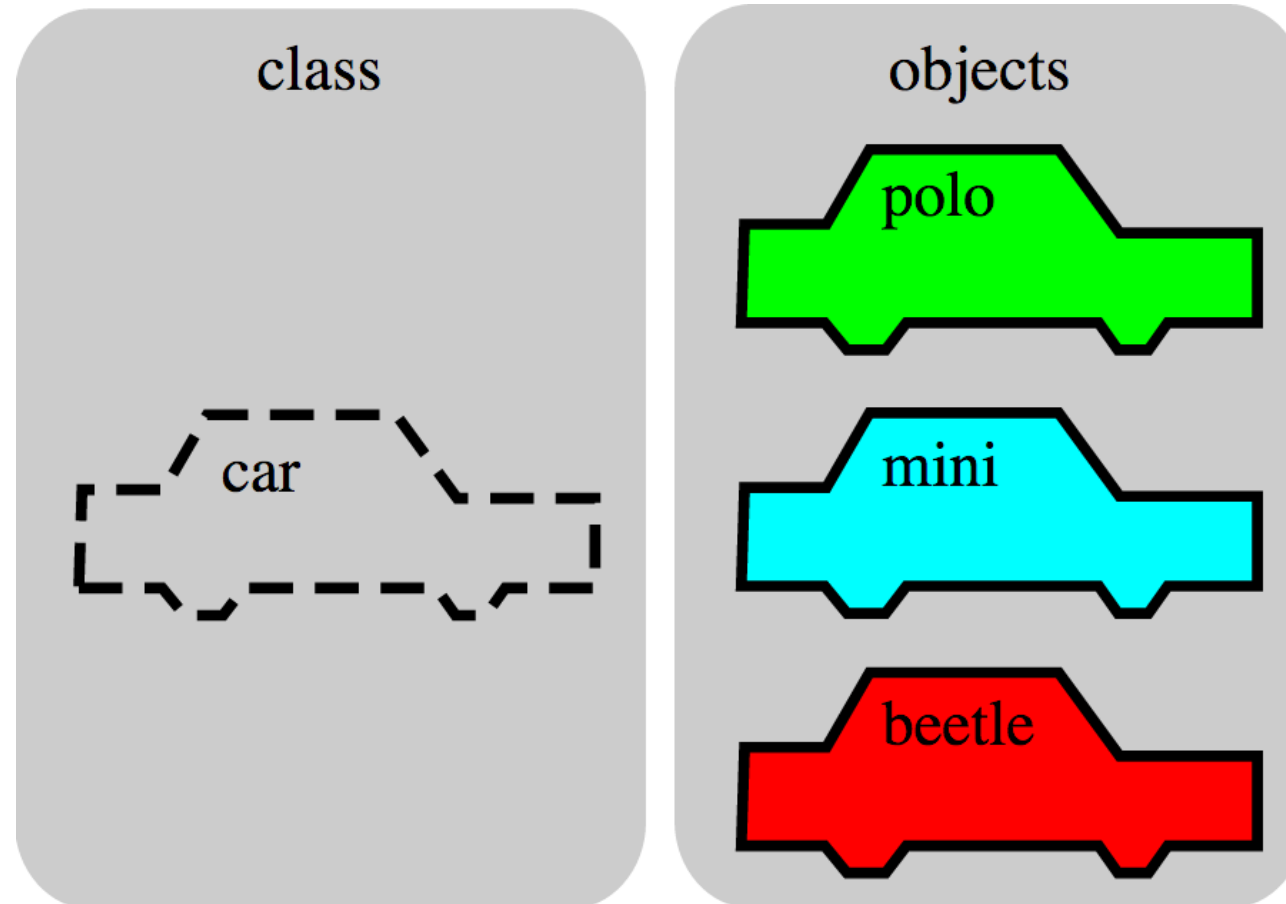
# Classes

- 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**
  - **Methods** are attributes that are functions

# 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>
```

# What's Going On?

- We initialize objects through constructors which return a new instance
  - `origin = Point(0, 0)`
  - `my_house = Point(5, 3)`
  - `campus = Point(8, 8)`
- We access attributes using 'dot notation'
  - `origin.x == 0`
  - `my_house.x == 5`
- We also call methods (functions) using dot notation:
  - `new_point = campus.subtract(my_house)`

# Computational Structures in Data Science

---

A Basic Bank Account

UC Berkeley



# 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

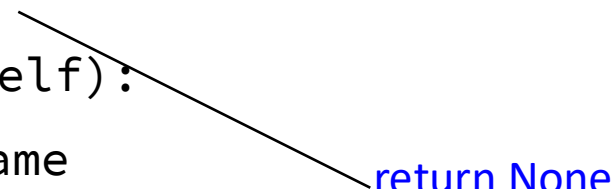
```
class BaseAccount:

    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

# Demo Where Does self come from?

- Python *binds* methods to each instance
- These two calls are the same:

```
my_account.withdraw(10)
```

```
BaseAccount.withdraw(my_account, 10)
```

# Check In:

```
cs88 = BaseAccount('CS88', 100)
```

```
cs88.withdraw(25)
```

```
cs88.balance -= 25
```

```
cs88.balanace()
```

```
# What's the value? Does it work?
```

# Check In:

```
cs88 = BaseAccount('CS88', 100)
```

```
c88c = BaseAccount('CS88', 100)
```

```
c88c == cs88
```

```
# What's the result? Does it work?
```

# 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 data is internal"
  - **`_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” (internal) 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
```

# Computational Structures in Data Science

---

Object-Oriented Programming:  
Class Attributes

UC Berkeley

# 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())
```

# Computational Structures in Data Science

---

Object-Oriented Programming:  
"Magic" Methods

UC Berkeley

# Learning Objectives

- Python's Special Methods define built-in properties
  - `__init__` # Called when making a new instance
  - `__sub__` # Maps to the `-` operator
  - `__str__` # Called when we call `print()`
  - `__repr__` # Called in the interpreter

# Special Initialization Method

`__init__` is called automatically when we write:

```
my_account = BaseAccount('me', 0)
```

```
class BaseAccount:
```

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

```
    def account_name(self):  
        return self.name
```

```
    def account_balance(self):  
        return self.balance
```

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

return None



# More special methods

```
class BaseAccount:
    ... (init, etc removed)
    def deposit(self, amount):
        self._balance += amount
        return self._balance

    def __repr__(self):
        return '< ' + str(self._acct_no) +
            '[' + str(self._name) + ']' >'

    def __str__(self):
        return 'Account: ' + str(self._acct_no) +
            '[' + str(self._name) + ']'

    def show_accounts():
        for account in BaseAccount.accounts:
            print(account)
```

# More Magic Methods

- We will **not** go through an exhaustive list!
- Magic Methods start and end with "double underscores" `__`
- They map to built-in functionality in Python. Many are logical names:
  - `__init__` → Class Constructor
  - `__add__` → + operator
  - `__sub__` → - operator
  - `__getitem__` → [] operator
  - `__repr__` and `__str__` → control output
- A longer list for the curious:
  - <https://docs.python.org/3/reference/datamodel.html>