Computational Structures in Data Science

Object-Oriented Programming (OOP)

Week 4, Summer 2024. 7/11 (Thurs)

Lecture 14

Berkeley
UNIVERSITY OF CALIFORNIA

Rev1: 2024-07-11 3:55 pm pst

Announcements

- Midterm next week!
- HW07, Lab07 out today (due: 7/15)
- Project01 (“Maps”) is ongoing.
  - isn’t k-means neat?

Midterm content

- Midterm will cover content from start of course up to (and including) OOP+Inheritance, aka:
  - Start (inclusive): Lecture 01: “Welcome & Intro” (6/17)
  - End (inclusive): Lecture 15: “OOP – Inheritance” (7/15)
- Midterm will be done through Zoom + Gradescope
- Study tip: past C88C exams can be found here: https://c88c.org/sp24/articles/resources.html#past-midterms
  - Take a look to get a sense of what C88C exams tend to look like. (I highly, highly encourage this)
    - “Be prepared” – Boy Scouts
    - “Luck is when preparation meets opportunity” – Roman philosopher Seneca

Midterm logistics

- The midterm will be held over Zoom + Gradescope
  - You must have your camera + screen sharing on during the entire exam, and we will be doing screen+camera recording.
- You must take the exam in a quiet room with no other students present
- Things to bring to the exam (and nothing else!):
  - **Photo ID**. Ideally your UCB student ID, but anything with your name + photo is fine, eg: Passport, driver's license, etc.
  - **(Optional)** Five (5) pages of handwritten (not typed!) notes
  - **(Optional, recommended)** Additional blank scratch paper, pencil/pen/eraser.
- We will provide everyone with a 1-2 page digital PDF of additional reference
- Other than the above notes, the exam will be closed book, closed notes.
- (For more info, stay tuned for an Ed post)

Computational Structures in Data Science

Object-Oriented Programming (OOP)

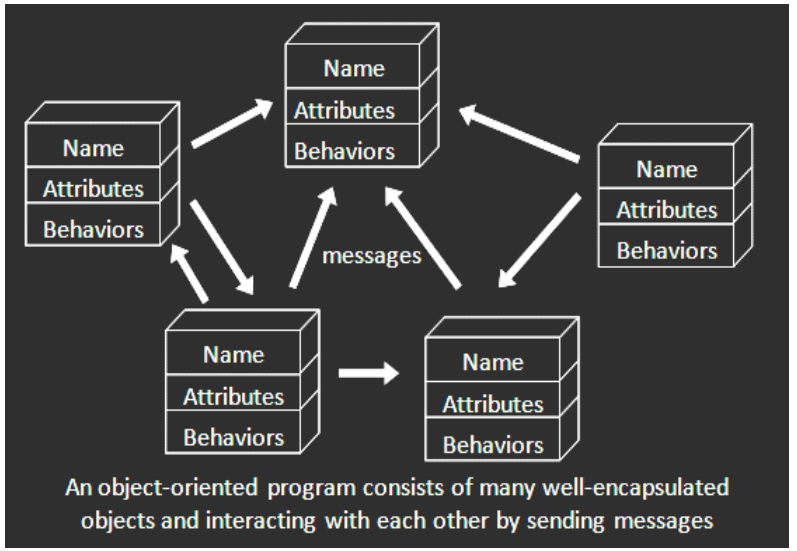
Berkeley
UNIVERSITY OF CALIFORNIA

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 cade 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,

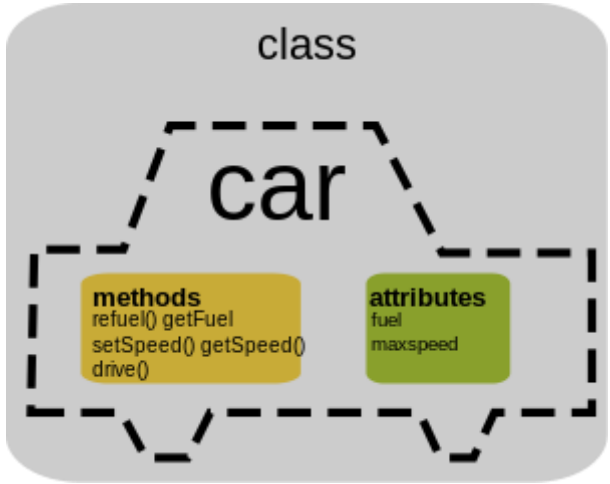
Turing Award Winner, UC Berkeley '61.

Full interview

Classes

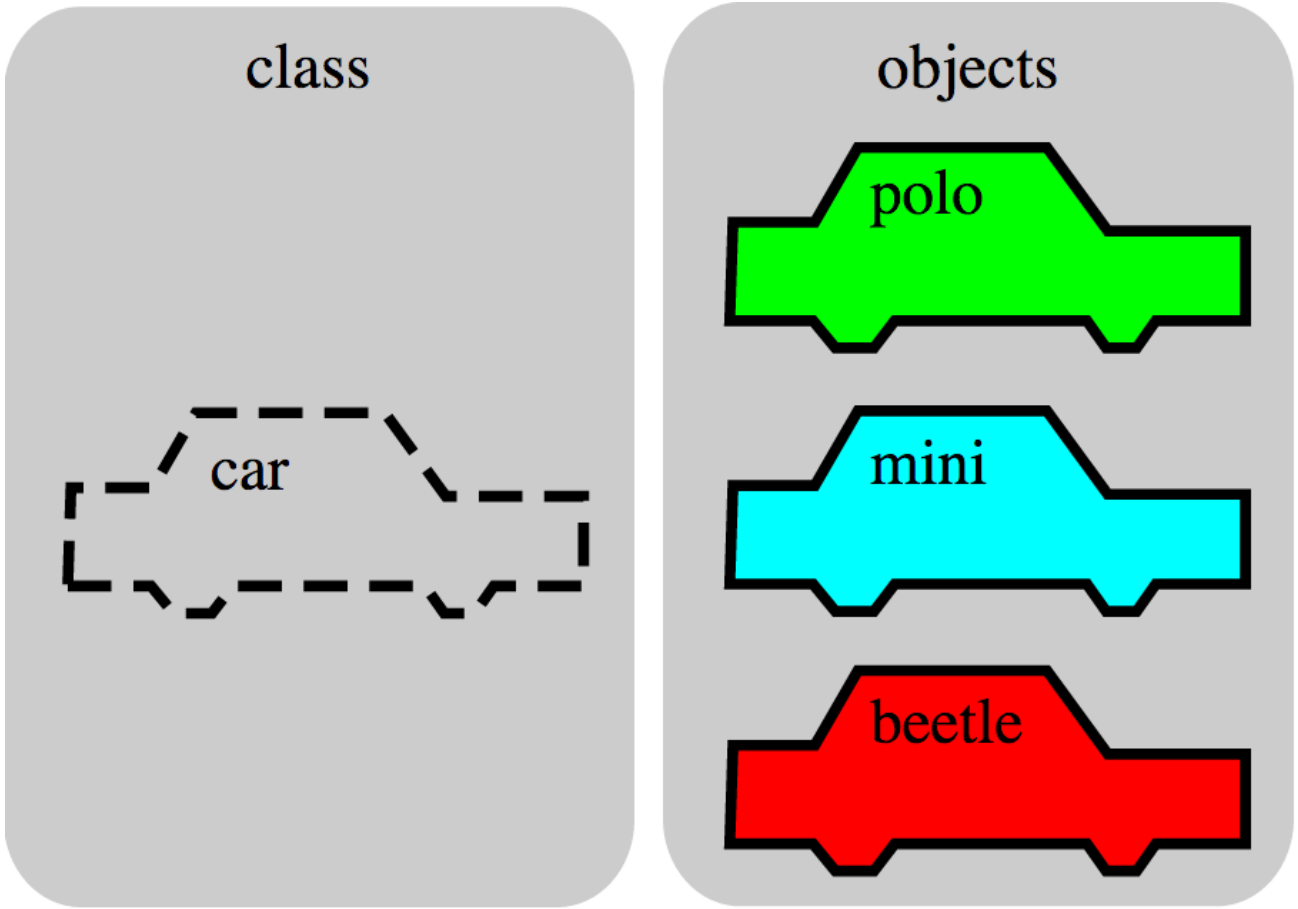
- Consist of data and behavior, bundled together to create abstractions
  - Abstract Data Types use functions to create abstractions
  - Classes define a new **type** in a programming language
    - They make the "abstract" data type concrete.
- A class has
  - attributes (variables)
  - methods (functions)

  that define its behavior.



Objects

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



Analogy:
A “class” is a “blueprint”

An “object” is an actual “built/instantiated” entity based off the blueprint (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>
```

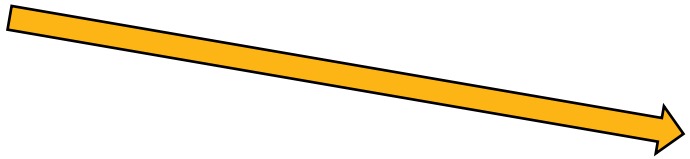
From ADTs to Classes

- Recall our Point ADT from earlier

```
# constructor
def create_point(x, y):
    return [x, y]

# selectors
def get_x(point):
    return point[0]
def get_y(point):
    return point[1]

# Operators
def distance_l2(p1, p2):
    # L2 distance btwn points p1, p2
    return ((get_x(p1) - get_x(p2))**2
+ (get_y(p1) - get_y(p2))**2)**0.5
```



```
class Point:
    # Constructor
    def __init__(self, x, y):
        # instance vars
        self.x = x
        self.y = y

    # Selectors
    def get_x(self):
        return self.x
    def get_y(self):
        return self.y

    # Instance Methods
    def dist_l2(self, pt_other):
        # L2 dist btwn myself (self) and pt_other
        return ((self.x - pt_other.x)**2
                + (self.y - pt_other.y)**2)**0.5
```

From ADTs to Classes (usage)

```
# constructor
def create_point(x, y):
    return [x, y]
# selectors
def get_x(point):
    return point[0]
def get_y(point):
    return point[1]
# Operators
def distance_l2(p1, p2):
    # L2 distance btwn points p1, p2
    return ((get_x(p1) - get_x(p2))**2
+ (get_y(p1) - get_y(p2))**2)**0.5
```

```
>>> pt1 = create_point(1, 1)
>>> pt2 = create_point(2, 3)
>>> distance_l2(pt1, pt2)
2.23606797749979
```

```
class Point:
    # Constructor
    def __init__(self, x, y):
        self.x = x  # instance vars
        self.y = y
    # Selectors
    def get_x(self):
        return self.x
    def get_y(self):
        return self.y
    # Instance Methods
    def dist_l2(self, pt_other):
        # L2 dist btwn myself (self) and pt_other
        return ((self.x - pt_other.x)**2
                + (self.y - pt_other.y)**2)**0.5
```

```
>>> pt1 = Point(1, 1)
>>> pt2 = Point(2, 3)
>>> pt1.dist_l2(pt2)
2.23606797749979
```

Very similar in spirit! Interesting.

From ADTs to Classes (usage)

```
# constructor
def create_point(x, y):
    return [x, y]
# selectors
def get_x(point):
    return point[0]
def get_y(point):
    return point[1]
# Operators
def distance_l2(p1, p2):
    # L2 distance btwn points p1, p2
    return ((get_x(p1) - get_x(p2))**2
+ (get_y(p1) - get_y(p2))**2)**0.5
```

```
class Point:
    # Constructor
    def __init__(self, x, y):
        self.x = x  # instance vars
        self.y = y
    # Selectors
    def get_x(self):
        return self.x
    def get_y(self):
        return self.y
    # Instance Methods
    def dist_l2(self, pt_other):
        # L2 dist btwn myself (self) and pt_other
        return ((self.x - pt_other.x)**2
                + (self.y - pt_other.y)**2)**0.5
```

```
>>> pt1 = create_point(1, 1)
>>> pt1
[1, 1]
>>> type(pt1)
<class 'list'>
```

Our pt1 instance has type Point!

More explicit than ADT's

```
>>> pt1 = Point(1, 1)
>>> pt1
<__main__.Point object at 0x000001A2A589FFD0>
>>> type(pt1)
<class '__main__.Point'>
```

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 get_balance(self):
        return self.balance

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

new namespace

Instance attributes

The object

Dot notation (“dot operator”)

Instance methods

Constructor: 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 get_balance(self):
        return self.balance

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

return None

```
# calling the constructor ("__init__()")
my_account = BaseAccount("bob", 100)
# access an instance attribute via "dot operator"
>>> my_account.name
bob
# call an instance method (also via dot operator)
>>> my_account.account_name()
bob
>>> my_account.withdraw(5)
95
>>> my_account.get_balance()
95
>>> my_account.balance
95
```

(Aside) Public vs private attributes

- Sometimes, it's useful to distinguish between "public" and "private" data.
  - Used to clarify to programmers how you class should be used.
  - In Python an `_` prefix of an attribute name means "this thing is private"
    - Example: "thing.secret" vs "thing._secret"
  - `_foo` and `__foo` do different things inside a class.
  - More for the curious.

(Aside) Public vs private attributes

```
class Lockbox:
    def __init__(self, secret_password):
        # Convention: "_" prefix means other programmers
        # shouldn't access this!
        self._secret_password = secret_password
    def unlock(self, password):
        if password == self._secret_password:
            return "Unlocked"
        else:
            return "Invalid password"

>>> my_lockbox = Lockbox("pull the lever")
# mwahaha, conventions won't stop me
>>> my_lockbox.unlock(my_lockbox._secret_password)
Unlocked
```

In other programming languages, like Java/C++, there is strict private/public accessibility syntax that the language will enforce. (CS 61B)

But Python doesn't support this, instead relies on convention and "disciplined developers" to not rely on "_VAR" that aren't part of the public API

Important: for this class we won't ask about public/private in exams.

Class variables vs instance variables

- Class variables vs Instance variables:
  - Class variable set for all instances at once
    - aka a “global var” for the class
  - Instance variables per instance value
    - aka a “local var” for each instance

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
```

```
>>> BaseAccount.account_number_seed
1000
>>> acc1 = BaseAccount("accountA", 100)
>>> BaseAccount.account_number_seed
1001
>>> acc2 = BaseAccount("accountB", 50)
>>> BaseAccount.account_number_seed
1002
# You can also access Class variables from
# instances, but for reasons*, in this
# class (and, IMO, beyond), please always
# access Class variables from the Class name
>>> acc1.account_number_seed
1002
>>> acc2.account_number_seed
1002
```

* (Optional) for reasons, read this SO post

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())
```

You can have class methods as well!

Note the missing `self`. This is because class methods aren't called on an instance, it's called on the Class itself!

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):
        return self._name
    def account_no(self):
        return self._acct_no
    def balance(self):
        return self._balance
    def show_accounts():
        for account in BaseAccount.accounts:
            print(account.name(),
                  account.account_no(),
                  account.balance())
```

```
>>> acc1 = BaseAccount("A", 100)
>>> acc2 = BaseAccount("B", 50)
>>> BaseAccount.show_accounts()
A 1000 100
B 1001 50
```

Note: since `show_accounts()` is a Class method (not an instance method), I call it on `BaseAccount.show_accounts()`, NOT `acc1.show_accounts()`

OOP Terminology (jargon). Any questions?

- There's a lot of synonyms in OOP. I'll enumerate the most common ones:
  - **Instance attributes**, aka: instance variables
  - **Instance methods**
  - **Class attributes**, aka: class variables
  - **Class methods**, aka: static methods
  - **Calling a class constructor**, aka: "instantiate an object", "create an instance of the class"
- In this class, I'll try to be consistent, but do get familiar with all of this jargon.

