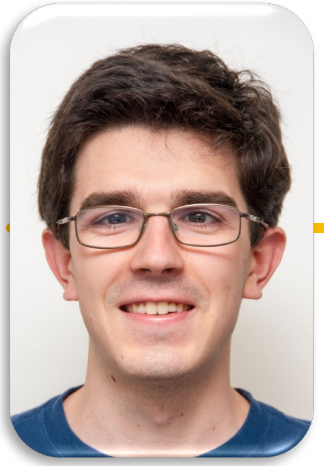


UC Berkeley EECS
Lecturer
Michael Ball

Computational Structures in Data Science

Object-Oriented Programming: Part 3, More OOP Practice

Announcements

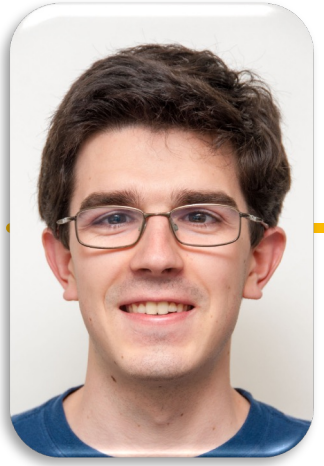
- Attendance: https://go.c88c.org/here
- Passcode: spring forward

- Class Chat: https://go.c88c.org/chat

Go to the Homework Party Thurs 7-9pm in Cory 293.

Announcements

- Midterm 3/21 7-9pm
  - Locations and assignments will be sent soon – please watch Ed.
    - » Students with DSP accommodations + students who filled out the remote/alternate form should have received emails.
  - Unlimited Handwritten Sheets – but try to use no more than 3-4!
- Hetal's Lecture: Mon 3/20 – Exam Review / Q&A
- Check the Calendar!
  - TA-led Topical Reviews (start today with HOFs at 4pm!)
  - Exam Review Sessions led by Tutors
  - CSM + HKN review sessions too
- No labs this week or next week – but we **will have lecture Weds 3/22!**
- TAKE A DEEP BREATH! Y'all can do this. ☺



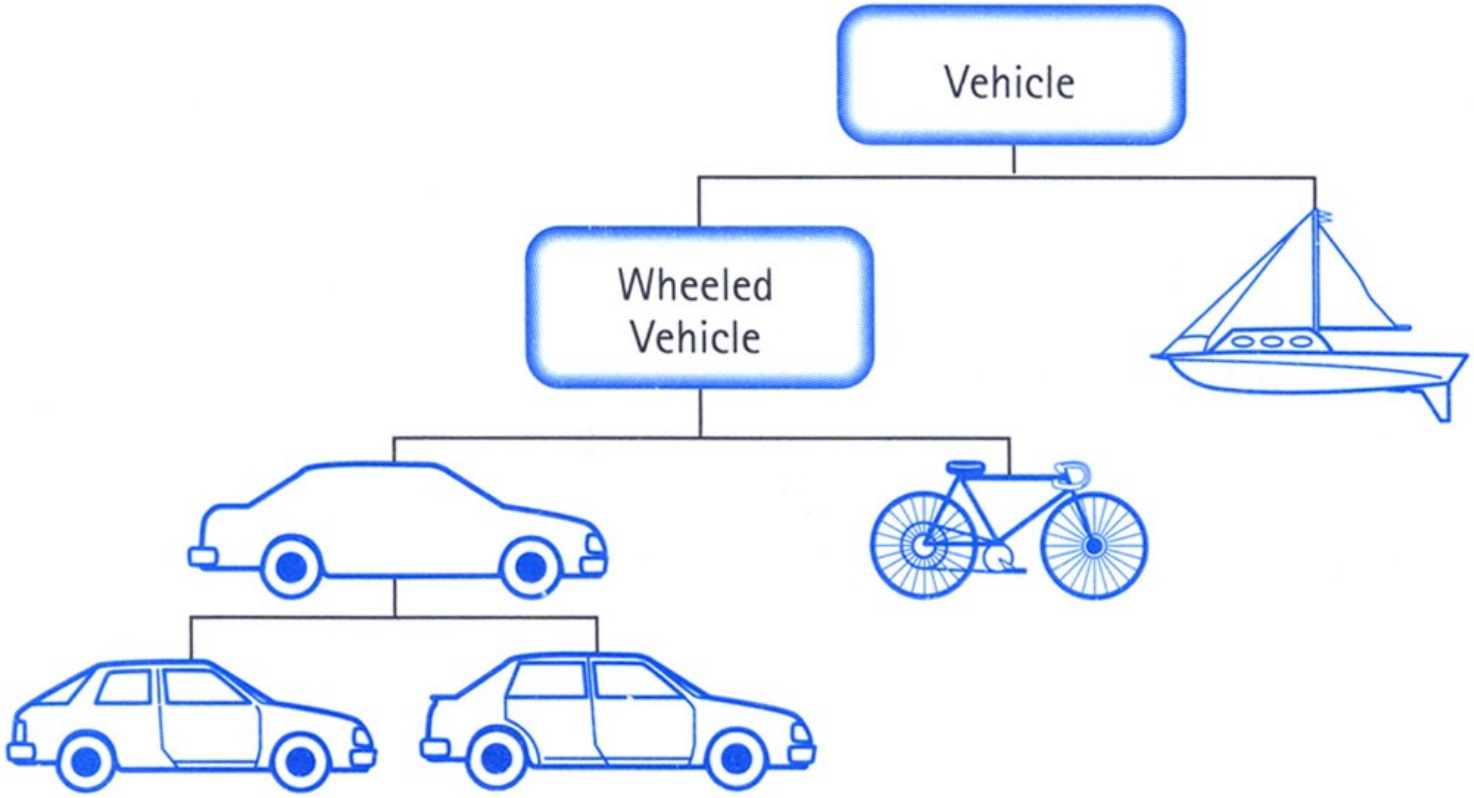
UC Berkeley EECS
Lecturer
Michael Ball

Computational Structures in Data Science

Object-Oriented Programming: Inheritance Review

Class Inheritance

- Classes can inherit methods and attributes from parent classes but extend into their own class.
- "is a" relationship



Example

```
class BaseAccount:
    def __init__(self, name, initial_deposit):
        # Initialize the instance attributes
        self._name = name
        self._acct_no = Account._account_number_seed
        Account._account_number_seed += 1
        self._balance = initial_deposit

class CheckingAccount(BaseAccount):
    def __init__(self, name, initial_deposit):
        # Use superclass initializer
        BaseAccount.__init__(self, name, initial_deposit)
        # Alternatively:
        # super().__init__(name, initial_deposit)
        # Additional initialization
        self._type = "Checking"
```

Accessing the Parent Class

- `super()` gives us access to methods in the parent or "superclass"
  - Can be called anywhere in our class
  - Handles passing `self` to the method
  - Handles looking up an attribute on a parent class, too.
- We can directly call `ParentClass.method(self, …)`
  - This is not quite as flexible if our class structure changes.

- In general, prefer using super()!
- Outside of C88C, things can get complex…
  - https://docs.python.org/3/library/functions.html#super

super() and Multiple Parent Classes

- In general, `super()` is "smart"
  - It tries to find the most correct parent class
  - Super will search through classes with multiple parent classes, or a long hierarchy of classes
- ParentClass is less flexible, but very specific.
  - Use it if you know you *always* want the same class to be used.

When Should You Use Inheritance?

Use inheritance to *refine* the behavior of a parent.

For example, our BaseAccount allows us to overdraft our account.

We might want to protect against this:

```
class CheckingAccount(BaseAccount):
    # (…omitted…(
    def withdraw(self, amount):
        if self.account_balance() - amount < 0:
            return "ERROR: You are not allowed to overdraft a
CheckingAccount."
        return super().withdraw(amount)
```

Inheritance & Class Attributes - Warning

Previously, we wrote something like this:

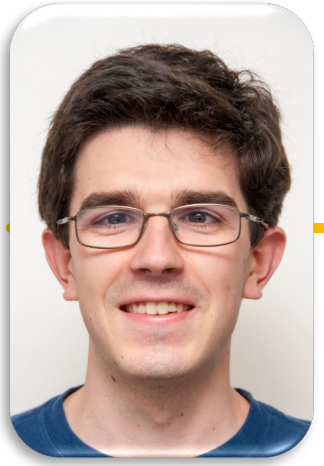
```
class SavingsAccount(BaseAccount):
    interest_rate = 0.02

    def accrue_interest(self):
        self._balance = self._balance * (1 +
SavingsAccount.interest_rate)
```

What happens when we have a new subclass?

```
class RetirementSavingsAccount(SavingsAccount):
    interest_rate = 0.05
```

Solution: use `self.interest_rate` instead, which will look up the appropriate attribute.



UC Berkeley EECS
Lecturer
Michael Ball

Computational Structures in Data Science

Object-Oriented Programming: Evolving The Bank Model

Keeping Track of Our Instances?

- **Problem:**
  - We can make many accounts… they all live in memory.
  - But how do we know what all of our accounts are?
  - How could we create an account number which is always increasing?
- Solution:
  - A *class* in Python can manage data shared across all instances

Composing Classes Together

- Currently, our BaseAccount stores a lot of data in class attributes…
  - Manages Account Numbers
  - Has a list of accounts
- This suggests we are trying to accomplish an entirely new kind of class, or object
  - A Bank!
- We should extract that these functions into their own class
- A bank can now manage:
  - making accounts
  - keeping track of account numbers
  - showing and listing accounts

Making A Bank

```
class Bank:
    def __init__(self, name, initial_account_number=1000):
        self.name = name
        self.__next_account_no = initial_account_number
        self.__accounts = []

    def new_account(self, name, initial_deposit=0,
account_type=CheckingAccount):
        account_no = self.__next_account_no
        account = account_type(name, initial_deposit, account_no, self)
        self.__next_account_no += 1
        self.__accounts.append(account)
        return account
```

Making New Accounts

- How do we control what try of account we want to have?
- We can use *higher order functions* to control which account type we create
- **`account_type`** becomes a function, because making a new class is a special kind of function

```
class Bank:
    def new_account(self, name, initial_deposit=0,
account_type=CheckingAccount):
        account = account_type(name, initial_deposit, account_no, self)
        return account
```

Live Demo

Should CheckingAccount inherit from Bank?

```
class Bank:
    def __init__(self, name, initial_account_number=1000):
        <. . . >


class CheckingAccount(Bank):
    def __init__(self, name, initial_deposit):
        super().__init__(name, initial_deposit)
        # Additional initialization
        self._type = "Checking"
```

- Generally speaking, no. A Bank does interact with account instances (e.g., stores accounts in a list) – but a Checking Account "is not a" Bank and should not have all the behaviors a Bank has.
- Classes can have many kinds of interactions with each other – not all of them should be implemented with inheritance!

Takeaways from OOP

- Syntax:
  - Define classes and instantiate objects
  - Access class and instance attributes, write and call methods
  - Understand look-up rules
- Inheritance:
  - What it allows you to do (access attributes/methods of parent classes)
  - How do you implement it in code
  - How to use super()

Takeaways from OOP

- Design:
  - Why would we want to create a class?
  - What are instance and class attributes, when would I implement something as one or the other?
  - When should classes inherit from each other?
  - Implement non-inheritance relationships between classes (like the relationship between Bank and CheckingAccount)