

Computational Structures in Data Science

OOP Part 3 Midterm Review



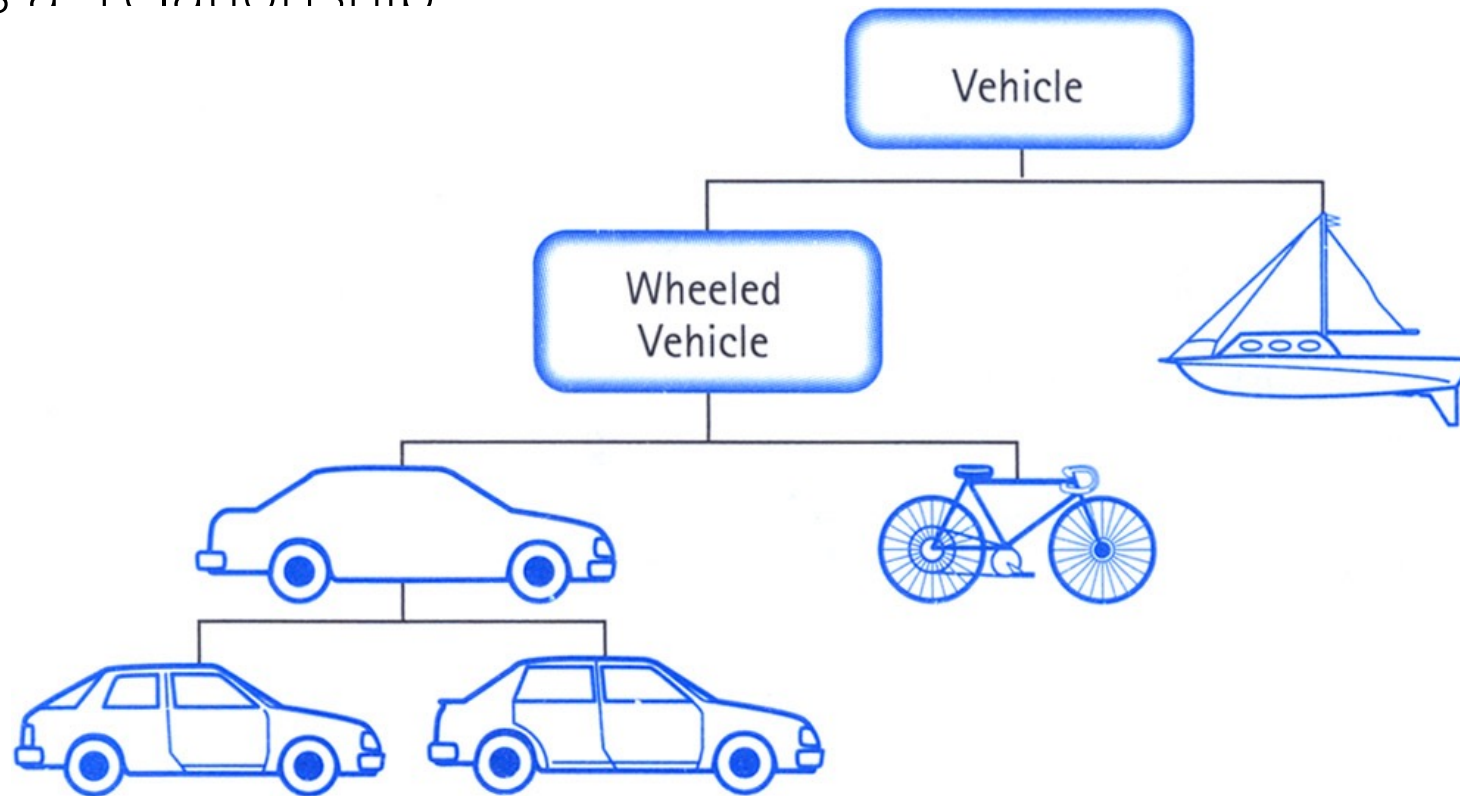
Computational Structures in Data Science

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.

Computational Structures in Data Science

Object-Oriented Programming: “Magic” Methods



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) + 'Goal: unambiguous'
            '[' + str(self._name) + ']' >'

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

    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>

Announcements & Policies

- Midterm:
 - 2 hours, 120 Minutes
 - Unlimited **Handwritten** Cheat sheets – More than ~3 is counter-productive
 - 1 CS88 Provided Reference Sheet
- Remember: HW6 / Lab 6 due next week, but are in scope.

Computational Structures in Data Science

Midterm Review



You are not your grades!
Do your best!

My Advice

- Don't rush!
 - **Slow is fast and fast is slow**
 - **BREATHE!**
- Skim the exam first
 - It's ok to do questions out of order!
 - Get the stuff you're good without out of the way
 - BUT don't spend too much time planning the exam.
- Read through the question once
 - What's it asking you to do at a high level?
 - What do the doctests suggest?
 - What techniques should you be using?
- **Use the scratch space!**

Midterm Topics

- Everything Through OOP w/ Inheritance
- Functions
- Higher Order Functions
 - Functions as arguments
 - Functions as return values
- Environment Diagrams
- Lists, Dictionaries
- List Comprehensions, Dictionary Comprehensions
- Abstract Data Types
- Recursion
- Object-Oriented Programming

Computational Structures in Data Science

Recursion Review



The Recursive Process

Recursive solutions involve two major parts:

- **Base case(s)**, the problem is simple enough to be solved directly
- **Recursive case(s)**. A recursive case has three components:
 - **Divide** the problem into one or more simpler or smaller parts
 - **Invoke** the function (recursively) on each part, and
 - **Combine** the solutions of the parts into a solution for the problem.

Recursion Key concepts – by example

1. Test for simple “base” case

2. Solution in simple “base” case

```
def sum_of_squares(n):  
    if n < 1:  
        return 0  
    else:  
        return sum_of_squares(n-1) + n**2
```



3. Assume recursive solution to simpler problem

4. “Combine” the simpler part of the solution, with the recursive case

Recursion With Lists

- Goal: Find the smallest item in a list, recursively.
- Consider: How do we break this task into smaller parts? What is the "smallest list"?
- We care about the size of the list itself, not the values.

```
def first(s):  
    """Return the first element in a sequence."""  
    return s[0]  
def rest(s):  
    """Return all elements in a sequence after the first"""  
    return s[1:]
```

```
def min_r(s):  
    '''Return minimum value in a sequence.'''  
    if   
    else:  
        
```

Computational Structures in Data Science

Questions from Ed

Berkeley
UNIVERSITY OF CALIFORNIA

Computational Structures in Data Science

Some Practice Questions



Exam Practice

- Spring 22 Q7
- Spring 20 Q5

7. (5.0 points) Closet Overhaul

You've designed a closet abstract data type to help you organize your wardrobe.

A closet contains two things:

- **owner**: the name of the closet owner represented as a string
- **clothes**: the collection of clothes in the closet represented as a dictionary, where the key is the clothing item name and the value is the number of times the clothing item has been worn.

The `make_closet` constructor takes in `owner` (a string) and `clothes` (a **list** of strings representing clothing items) and returns a closet ADT.

Given this, you've implemented the abstract data type as follows:

```
def make_closet(owner, clothes):  
    """ Create and returns a new closet. """  
    clothes_dict = {}  
    for item in clothes:  
        clothes_dict[item] = 0  
    return (owner, clothes_dict)  
  
def get_owner(closet):  
    """ Returns the owner of the closet """  
    return closet[0]  
  
def get_clothes(closet):  
    """ Returns a dictionary of the clothes in the closet """  
    return closet[1]
```

Given the closet ADT, implement the functions `wear_clothes` and `favorite_clothing_item`. You may not need all the lines provided, and you may need to change the indentation for some lines.

5. (10 points) Atey Ate Already

It's a lot more fun to think about food than take midterms, so let's look at the cheapest places to fulfill an order. Given the function `total_cost` and assuming it works as described, fill out `find_restaurant` to find the cheapest restaurant to fulfill the order.

Remember: Pay close attention to the doctests to guide your solution.

```
def total_cost(restaurant, order):
    """
    Function that returns the total cost of an order at a certain
    restaurant. Returns -1 if fulfilling the request is not possible.
    >>> total_cost('chipotle', ['burrito', 'taco'])
    11.96
    >>> total_cost('sliver', ['boba'])
    -1.0
    """
    # We have omitted how this function works.

def find_restaurant(restaurants, order):
    """
    Function that returns the cheapest restaurant and price as the first
    element of a list followed by the prices for each of the restaurants.
    In the case that no restaurant can fulfill the order, the first
    element should be ['None found!', -1]. In the case that two
    restaurants have the same price, keep the first restaurant.
    Hint: Use total_cost!
    >>> find_restaurant(['chipotle', 'la burrita'], ['burrito', 'taco'])
    [['la burrita', 9.78], [['chipotle', 11.96], ['la burrita', 9.78]]

    >>> find_restaurant(['sliver', 'cheeseboard'], ['boba'])
    [[None found!, -1.0], ['sliver', -1.0], ['cheeseboard', -1.0]]
    """
```

SP20 #6

(10 points) Rooms within Rooms within Rooms

You are a Data Scientist hired by UC Berkeley to find the largest room on campus. In order to schedule midterms, your job is return the room and its capacity. The data on all the rooms plus capacity is in a weird format of three element lists, where the first element is the room, the second element is the capacity, and the third element is either the rest of the data or None. Assume that the capacity of each of the rooms is unique.

That is, the data look like ['Room', Number, [...]].

Use the following lines of code to fill in the body of the function. You will need to fill in the blanks of the lines provided. Some lines are optional.

```
return [rooms[0], rooms[1]]
largest_left = find_largest(_____)
if rooms[2] == ____:
    if largest_left[1] > rooms[1]:
        return _____
    return _____
else: # this line is optional, depending upon your solution
else: # this line is optional, depending upon your solution
```

```
def find_largest(rooms):
    """
    Return the largest room from a weirdly nested list.
    You can assume rooms is always 3 items long.
    >>> rooms = ['Evans', 150, ['Wheeler', 700, ['Stadium', 50000, None]]]
    >>> find_largest(rooms)
    ['Stadium', 50000]
    >>> find_largest(['Evans', 150, None])
    ['Evans', 150]
    """
```

```
def find_largest(rooms):
    """
    Return the largest room from a weirdly nested list.
    You can assume rooms is always 3 items long.
    >>> rooms = ['Evans', 150, ['Soda', 200, ['Wheeler', 700, ['Stadium', 50000, None]
    >>> find_largest(rooms)
    ['Stadium', 50000]
    >>> find_largest(['Evans', 150, ['Hearst Annex', 50, None]])
    ['Evans', 150]
    """
    if rooms[2] == None:
        return [rooms[0], rooms[1]]
    largest_left = find_largest(rooms[2])
    if largest_left[1] > rooms[1]:
        return largest_left
    return [rooms[0], rooms[1]]
```



- i. (4.0 pt) Implement the `find_new_interest` method that returns a string representing a new potential interest for this `User`. To determine this new interest, first identify this user's most similar follower that has the largest number of mutual interests with this user. Then return a randomly selected interest from this follower. But be careful, this randomly selected interest must not already exist in this user's interests (otherwise it would not be new!).

For this problem, assume that the user's interests and followers are non-empty. Note that the `separate_interests` function (see `User` class skeleton) may be helpful here. You may use `random.choice(lst)` to return a randomly selected item from a list, `lst`.

```
def find_new_interest(self):
    """
    >>> u1 = User('bob', ['cooking', 'archery', 'tv'])
    >>> u2 = User('alice', ['shopping', 'guitar', 'cooking']) # has one in common with bob
    >>> u3 = User('mike', ['poker', 'tv', 'cooking']) # has two in common with bob
    >>> u1.add_follower(u2)
    >>> u1.add_follower(u3)
    >>> u1.find_new_interest()
    'poker'
    """
    most_similar_follower = max(
        -----,
        key = -----
    )
    return random.choice(-----)
```

Write the fully *completed* `find_new_interest` function below using the skeleton code provided. You may not add, change, or delete lines from the skeleton code.



SP20 #5

(10 points) Atey Ate Already

It's a lot more fun to think about food than take midterms, so let's look at the cheapest places to fulfill an order. Given the function `total_cost` and assuming it works as described, fill out `find_restaurant` to find the cheapest restaurant to fulfill the order.

Remember: Pay close attention to the doctests to guide your solution.

```
def total_cost(restaurant, order):
    """
    Function that returns the total cost of an order at a certain
    restaurant. Returns -1 if fulfilling the request is not possible.
    >>> total_cost('chipotle', ['burrito', 'taco'])
    11.96
    >>> total_cost('sliver', ['boba'])
    -1.0
    """
    # We have omitted how this function works.

def find_restaurant(restaurants, order):
    """
    Function that returns the cheapest restaurant and price as the first
    element of a list followed by the prices for each of the restaurants.
    In the case that no restaurant can fulfill the order, the first
    element should be ['None found!', -1]. In the case that two
    restaurants have the same price, keep the first restaurant.
    Hint: Use total_cost!
    >>> find_restaurant(['chipotle', 'la burrita'], ['burrito', 'taco'])
    [['la burrita', 9.78], [['chipotle', 11.96], ['la burrita', 9.78]]

    >>> find_restaurant(['sliver', 'cheeseboard'], ['boba'])
    [[None found!, -1.0], ['sliver', -1.0], ['cheeseboard', -1.0]]
    """
```

```

def findRestaurant(restaurants, order):
    """
    Function that returns the cheapest restaurant and price as the first
    element of a list followed by the prices for each of the restaurants.
    In the case that no restaurant can fulfill the order, the first
    element should be ['None found!', -1].
    Hint: Use totalCost! In the case that two restaurants have the same price,
    keep the first restaurant.
    >>> findRestaurant(['chipotle', 'la burrita'], ['burrito', 'taco'])
    [['la burrita', 9.78], [['chipotle', 11.96], ['la burrita', 9.78]]
    >>> findRestaurant(['sliver', 'cheeseboard'], ['boba'])
    [[None found!, -1.0], ['sliver', -1.0], ['cheeseboard', -1.0]]
    """

    placesList = [ [restaurant, totalCost(restaurant, order)]
                    for restaurant in restaurants ]

    minCost = -1.0
    cheapestPlace = "None found!"
    for place in range(placesList):
        if place[1] != -1.0 and (place[1] < minCost or minCost == -1.0):
            minCost = place[1]
            cheapestPlace = place[0]
    return [cheapestPlace, minCost] + placesList

```