Computational Structures in Data Science

Object-Oriented Programming: Part 2, Inheritance

Week 5, Summer 2024. 7/15 (Mon)

Lecture 15

Berkeley
UNIVERSITY OF CALIFORNIA

Announcements

- Midterm this week!
- Project01 (“Maps”) due 7/18 (Thurs!)
- HW07, Lab07 due tonight!

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)

Today’s lecture content

- OOP: inheritance
- Python “magic methods”
  - ex: __init__, __add__, __repr__, __str__, etc

Computational Structures in Data Science

Reviewing Our Account

Berkeley
UNIVERSITY OF CALIFORNIA

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
```

Python's Instance Attributes

- `self.attribute_name = x`
  - Sets up an attribute which can be modified by anyone

(Optional) internal/private Instance Attributes

- `self._attribute_name = x`
  - Sets up an attribute which is suggested that is "internal only"
  - e.g. `my_instance._attribute_name = y` will work, but should *look* wrong.
  - Internally, `self._attribute_name = y` is OK.
- `self.__attribute_name = x`
  - Sets up an attribute which is *private*
  - e.g. `my_instance.__attribue_name = y` will *error!*
  - Internally, `self.__attribute_name = y` is OK.

Important: in this class, we will not test you on internal/private instance vars, eg "_VAR" vs "__VAR"

Class Attributes: 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
  - We call these *class attributes* which are distinguished from i*nstance attributes*

Classes Can Have Attributes Too!

- Class attributes (as opposed to *instance* attributes) belong to the class itself, instead of each object
  - This means there is one value which is shared for all of the class's objects
- Be Careful!
  - It's easy to overdo class attributes

- Methods that rely only on class attributes are called *class methods*
  - Python has some special features we won't use, but are useful
  - Declaring a method as belonging to a class, not an instance.

Important: in this class, we will not require you to use @classmethod, @staticmethod in exams, nor do you need to understand the difference between the two. But if you're curious, here's a good explanation of the difference.

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())
```

Are There Better Approaches?

- BEWARE! Class attributes are useful but can get confusing.
- Perhaps what want is a **`Bank()`** class
  - The bank would have a create_account() method
  - Each Bank() would have its own accounts list, as a set of instance variables.

```
class Bank():
    def __init___(self):
       self.account_no_seed = 1000
       self.accounts = []
   def create_account(self, name, balance):
       acct = BaseAccount(name, balance, self.account_no_seed)
       self.accounts.append(acct)
       self.account_no_seed += 1
```

Computational Structures in Data Science

Object-Oriented Programming: Inheritance

Berkeley
UNIVERSITY OF CALIFORNIA

Learning Objectives

- Inheritance allows classes to reuse methods and attributes from a parent class.
- super() is a new method in Python
- Subclasses or child classes are distinct from another, but share properties of the parent.

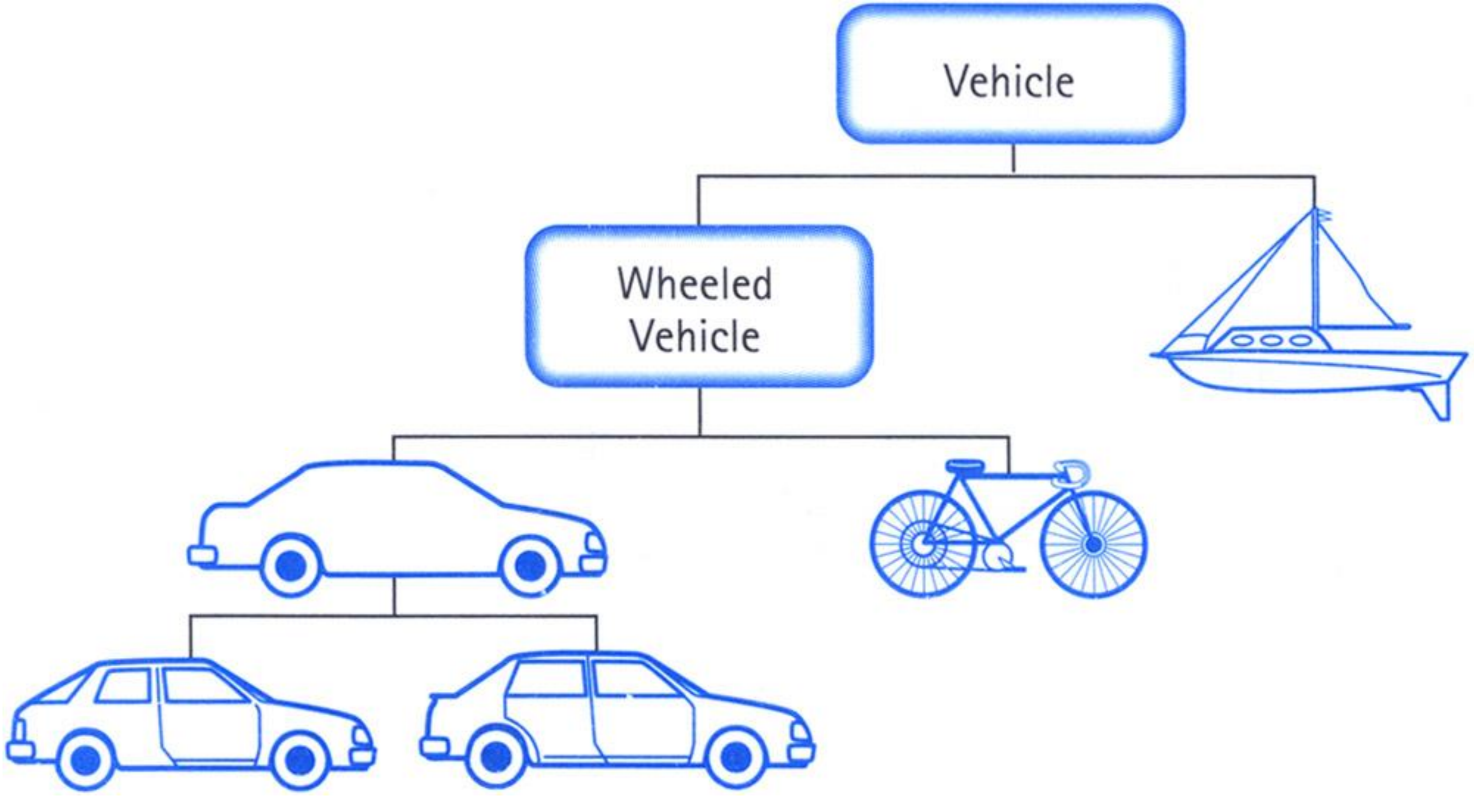
Class Inheritance: Motivation

- Say we are working in the vehicle domain, and define a class for each vehicle type
- Observation: many of these classes are very similar
  - Car, SportsCar, SUV have lots of shared functionality, eg methods like: `drive(), fill_up_gas(), open_door()`, etc
- However, they are all different classes, so we will likely have lots of repeated code
- Is there a better way?

```
class Car:
    pass

class SportsCar:
    pass

class SUV:
    pass

class Tank:
    pass

class Boat:
    pass

...
```

Class Inheritance

- **Idea**: model our vehicle classes via a type hierarchy!
- Classes can inherit methods and attributes from parent classes but extend into their own class.



Inheritance

- Define a class as a specialization of an existing class
- Inherent its attributes, methods (behaviors)
- Add additional ones
- Redefine (specialize) existing ones
  - Ones in superclass still accessible in its namespace

Python class statement

```
class ClassName:
    <statement-1>
    .
    .
    .
    <statement-N>
```

```
class ClassName ( inherits / parent-class ):
    <statement-1>
    .
    .
    .
    <statement-N>
```

Example

CheckingAccount inherits from the BaseAccount class

BaseAccount is the “parent class” of the CheckingAccount class.

(jargon) CheckingAccount “extends” BaseAccount

```
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 (recommended):
        # super().__init__(name, initial_deposit)
        # Additional initialization
        self._type = "Checking"
```

Accessing the Parent Class: super()

- `super()` *binds* methods in the parent or "superclass" to the current instance
  - 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

Accessing the Parent Class: super()

```python
class Person:
    def __init__(self, age):
        self.age = age
    def have_birthday(self):
        self.age += 1
        return f"Now I'm one year older: {self.age}!"
    def have_fun(self):
        return "Whee!"


class Employee(Person):
    def __init__(self, age, company_name):
        super().__init__(age)
        self.company_name = company_name
    def have_birthday(self):
        out_super = super().have_birthday()
        return out_super + f" Well, time for work at {self.company_name}!"
    def have_fun(self):
        return f"I can't have fun, I have to work at {self.company_name}!"
```

```
>>> youngster = Person(10)
>>> youngster.have_birthday()
Now I'm one year older: 11!
>>> youngster.have_fun()
Whee!
>>> employee = Employee(35, "BigCorp")
>>> employee.have_birthday()
Now I'm one year older: 36! Well, time
for work at BigCorp!
>>> employee.have_fun()
I can't have fun, I have to work at
BigCorp!
```

Computational Structures in Data Science

Object-Oriented Programming: Evolving The Bank Model

Berkeley
UNIVERSITY OF CALIFORNIA

Composing Classes Together

- Currently, our `BaseAccount` stores a lot of data in class attributes…
- 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

Demo: lecture15.py, BaseAccount + Bank class

Computational Structures in Data Science

Object-Oriented Programming: "Magic" Methods

Berkeley
UNIVERSITY OF CALIFORNIA

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
```

More special methods: __repr__ vs __str__

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

Goal: unambiguous

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

Goal: human readable

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

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

More special methods: __repr__ vs __str__

```
class BaseAccount:
    ...
    # Display representation
    def __repr__(self):
        return f'<{self.account_type()}:
{self.account_name()}-{self.account_number()}>'

    # Print representation
    def __str__(self):
        return f'{self.account_type()}:
{self.account_name()}-{self.account_number()}
Balance: {self._balance}'
```

__repr__ goal: unambiguous

__str__ goal: human readable

```
# Tip: __repr__ vs __str__

# Python interpreter outputs repr()
>>> account_c
<BaseAccount: account_c-1000>
# print() calls str()
>>> print(account_c)
BaseAccount: account_c-1000 Balance:
9999
>>> str(account_c)
'BaseAccount: account_c-1000
Balance: 9999'
>>> repr(account_c)
'<BaseAccount: account_c-1000>'
```

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

Live Demo

Demo: lecture15.py, magic methods

Aside: opinions on OOP

- Object oriented programming (OOP) got really popular in the 1980's/1990's.
  - Java ("what if EVERYTHING was a Class?"), C++ ("C with Classes")
- With hindsight, one learning is that OOP is not always the software paradigm
- OOP is a great tool…for the right situation
  - some people have very strong opinions for and against OOP
- **Alternatives**: functional (aka map/reduce/filter), imperative, declarative (SQL)
- **My advice**: try to use the best tool for the problem at hand.
  - Avoid the "with a hammer, every problem looks like a nail" syndrome

Aside: opinions on coding. Any questions?

- At the end of the day: it's very hard to design programs the "right way"
  - A "good design" lets you be productive and solve problems. Feels great!
  - A "bad design" feels like you are suffocated by an unwieldy API, etc
- This is an art! Like any craft, to get better at it **you must practice**
  - Experience + wisdom. (usually gained by learning through mistakes, heh)
- **"Your first 100 songs will suck. So, start writing and get them out of the way!"** – advice on songwriting
  - 100% the same for writing code!

Source: https://www.scotthyoung.com/blog/2019/08/26/better-writing-brainstorm/