Computational Structures in Data Science

Object-Oriented Programming: Attributes

Berkeley
UNIVERSITY OF CALIFORNIA

Announcements

- Midterm Grades are out!
- Good regrade requests:
  - Please be polite.
  - Please show that you tested your answer in a terminal and why it should be correct within the rubric.

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
- `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.__attribute_name = y` will *error!*
  - Internally, `self.__attribute_name = y` is OK.

Computational Structures in Data Science

Object-Oriented Programming: Class Attributes

Berkeley
UNIVERSITY OF CALIFORNIA

Class Attributes: Keeping Track of Our Instances?

- Problem:
  - We can make many objects… they all live in memory.
  - But how do we keep track of all accounts and easily see what all of our accounts are?
  - How could we create an account number which is always increasing?
  - How can we manage data that needs to be shared across all accounts?
- 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.

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())
```

Live Demo

Are There Better Approaches?

- BEWARE! Class attributes are useful but can get confusing.
- Perhaps what we 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: "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
```

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)
```

Goal: unambiguous

Goal: readable

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

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 on another, but share properties of the parent.

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

Class Inheritance

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