CS229 Python Tutorial

TA: Mario Srouji
Python basics demo
Python OOP
Why Classes?

- Logical grouping of data and functions (which are called methods)
- We try to create classes with logical connections or unified functionality
- Modeling technique, a way of thinking about programs
- Very useful for maintaining “state” in programs
- Think of a class as a sort of “blueprint”
class Customer(object):
    """A customer of ABC Bank with a checking account. Customers have the following properties:

    Attributes:
    - name: A string representing the customer's name.
    - balance: A float tracking the current balance of the customer's account.
    """

    def __init__(self, name, balance=0.0):
        """Return a Customer object whose name is *name* and starting balance is *balance*."
        self.name = name
        self.balance = balance

    def withdraw(self, amount):
        """Return the balance remaining after withdrawing *amount* dollars."
        if amount > self.balance:
            raise RuntimeError('Amount greater than available balance."
        self.balance -= amount
        return self.balance

    def deposit(self, amount):
        """Return the balance remaining after depositing *amount* dollars.""
        self.balance += amount
        return self.balance
Class Instantiation

• The **class Customer(object) line does not “create” the class - this is defining the “blueprint”**

• To instantiate the class - we call the **\_init\_** method with the proper number of arguments (minus self)

  • \_init\_(self, name, balance=0.0)

• **mario = Customer(“Mario Srouji”, 1000.0) - instantiates an object mario of the class Customer**
class Customer(object):
    """A customer of ABC Bank with a checking account. Customers have the following properties:

Attributes:
    name: A string representing the customer's name.
    balance: A float tracking the current balance of the customer's account.
"""

    def __init__(self, name, balance=0.0):
        """Return a Customer object whose name is *name* and starting balance is *balance*."""
        self.name = name
        self.balance = balance

    def withdraw(self, amount):
        """Return the balance remaining after withdrawing *amount* dollars."""
        if amount > self.balance:
            raise RuntimeError('Amount greater than available balance.')
        self.balance -= amount
        return self.balance

    def deposit(self, amount):
        """Return the balance remaining after depositing *amount* dollars."""
        self.balance += amount
        return self.balance
What does `self` mean?

- `self` is the instance of the class we are using.
- When defining a function (method) inside of a class - need to include `self` as first argument so we can use it.
- Syntactical way to define that this particular method should be applied to the given object instance.
- `mario.withdraw(100.0) = Customer.withdraw(mario, 100.0)`
class Customer(object):
    """A customer of ABC Bank with a checking account. Customers have the
    following properties:

    Attributes:
        name: A string representing the customer's name.
        balance: A float tracking the current balance of the customer's account.
    """

def __init__(self, name, balance=0.0):
    """Return a Customer object whose name is *name* and starting
    balance is *balance*."
    self.name = name
    self.balance = balance

def withdraw(self, amount):
    """Return the balance remaining after withdrawing *amount* dollars.""
    if amount > self.balance:
        raise RuntimeError('Amount greater than available balance.')
    self.balance -= amount
    return self.balance

def deposit(self, amount):
    """Return the balance remaining after depositing *amount* dollars.""
    self.balance += amount
    return self.balance
What does `__init__` do?

• When we call `__init__` we are creating the object instance

• It is the class “constructor”

• To call the `__init__` method of a class, instantiate the class name with the arguments defined in `__init__`

• `mario = Customer(“Mario Srouji”, 1000.0)`

• Variables or “attributes” defined in the `__init__` method can be accessed inside and outside the class

• `mario.name = “bob”` will modify the name attribute
class Customer(object):
    """A customer of ABC Bank with a checking account. Customers have the following properties:

    Attributes:
    name: A string representing the customer's name.
    balance: A float tracking the current balance of the customer's account.
    """

    def __init__(self, name, balance=0.0):
        """Return a Customer object whose name is *name* and starting balance is *balance*."""
        self.name = name
        self.balance = balance

    def withdraw(self, amount):
        """Return the balance remaining after withdrawing *amount* dollars.""
        if amount > self.balance:
            raise RuntimeError('Amount greater than available balance."
        self.balance -= amount
        return self.balance

    def deposit(self, amount):
        """Return the balance remaining after depositing *amount* dollars.""
        self.balance += amount
        return self.balance
```python
class Customer(object):
    """A customer of ABC Bank with a checking account. Customers have the following properties:

    Attributes:
    name: A string representing the customer's name.
    balance: A float tracking the current balance of the customer's account.
    """

def __init__(self, name):
    """Return a Customer object whose name is *name*."
    self.name = name

def set_balance(self, balance=0.0):
    """Set the customer's starting balance."""
    self.balance = balance

def withdraw(self, amount):
    """Return the balance remaining after withdrawing *amount* dollars."""
    if amount > self.balance:
        raise RuntimeError('Amount greater than available balance.')
    self.balance -= amount
    return self.balance

def deposit(self, amount):
    """Return the balance remaining after depositing *amount* dollars."""
    self.balance += amount
    return self.balance
```
Good practice

- Looked reasonable - calling the `set_balance` method before using the instance of the class
- No way to communicate this to the user
- We can not force caller to invoke `set_balance`
- Rule of thumb - do not introduce an attribute outside of the `__init__` method
Instance Methods

- Function defined in a class is called a “method”
- Methods have access to all data contained in the instance of the object
- Can access and modify anything previously defined on self
- Since they use self, they require an instance of the class to be used - hence we call them instance methods
class Customer(object):
    """A customer of ABC Bank with a checking account. Customers have the following properties:

Attributes:
    name: A string representing the customer's name.
    balance: A float tracking the current balance of the customer's account.
"""

def __init__(self, name, balance=0.0):
    """Return a Customer object whose name is *name* and starting balance is *balance*."""
    self.name = name
    self.balance = balance

def withdraw(self, amount):
    """Return the balance remaining after withdrawing *amount* dollars."""
    if amount > self.balance:
        raise RuntimeError('Amount greater than available balance."
    self.balance -= amount
    return self.balance

def deposit(self, amount):
    """Return the balance remaining after depositing *amount* dollars."""
    self.balance += amount
    return self.balance
Static Methods

- Do not have access to `self`
- Work without requiring an instance to be present
- Do not have a `self` parameter
Inheritance

- The process in which a “child” class derives data and behavior from a “parent” class
- Avoids duplication of code - example in a moment
- Allows for creation of “abstract” classes - general templates
- Can use “abstract” classes to define specific instances depending on application
Why inheritance?

```python
class Car(object):
    ""
    A car for sale by Jeffco Car Dealership.
    ""
    Attributes:
    wheels: An integer representing the number of wheels the car has.
    miles: The integer number of miles driven on the car.
    make: The make of the car as a string.
    model: The model of the car as a string.
    year: The integer year the car was built.
    sold_on: The date the vehicle was sold.
    ""
    def __init__(self, wheels, miles, make, model, year, sold_on):
        """Return a new Car object.""
        self.wheels = wheels
        self.miles = miles
        self.make = make
        self.model = model
        self.year = year
        self.sold_on = sold_on

    def sale_price(self):
        """Return the sale price for this car as a float amount.""
        if self.sold_on is not None:
            return 0.0  # Already sold
        return 5000.0 * self.wheels

    def purchase_price(self):
        """Return the price for which we would pay to purchase the car.""
        if self.sold_on is None:
            return 0.0  # Not yet sold
        return 8000 - (.10 * self.miles)
    ...
```
Why inheritance?

class Truck(object):
    """A truck for sale by Jeffco Car Dealership.

Attributes:
    wheels: An integer representing the number of wheels the truck has.
    miles: The integral number of miles driven on the truck.
    make: The make of the truck as a string.
    model: The model of the truck as a string.
    year: The integral year the truck was built.
    sold_on: The date the vehicle was sold.
    """

    def __init__(self, wheels, miles, make, model, year, sold_on):
        """Return a new Truck object."""
        self.wheels = wheels
        self.miles = miles
        self.make = make
        self.model = model
        self.year = year
        self.sold_on = sold_on

    def sale_price(self):
        """Return the sale price for this truck as a float amount."""
        if self.sold_on is not None:
            return 0.0  # Already sold
        return 5000.0 * self.wheels

    def purchase_price(self):
        """Return the price for which we would pay to purchase the truck."""
        if self.sold_on is None:
            return 0.0  # Not yet sold
        return 10000 - (.10 * self.miles)

...
Why inheritance?

• The Car and Truck classes are almost identical - unnecessary duplication of code

• They share a lot of data and functionality in common

• Why not introduce an abstraction that allows us to combine these two Vehicle classes
Abstract classes

- The **Vehicle** class is a concept that allows us to embody reusable information.
- We can make the **Car** and **Truck** classes inherit from the **Vehicle** class.
- Let’s look at the example on the next slide.
Abstract classes

class Vehicle(object):
    """A vehicle for sale by Jeffco Car Dealership."

    Attributes:
        wheels: An integer representing the number of wheels the vehicle has.
        miles: The integral number of miles driven on the vehicle.
        make: The make of the vehicle as a string.
        model: The model of the vehicle as a string.
        year: The integral year the vehicle was built.
        sold_on: The date the vehicle was sold.
    """

    __metaclass__ = ABCMeta

    base_sale_price = 0
    wheels = 0

    def __init__(self, miles, make, model, year, sold_on):
        """""""
        self.miles = miles
        self.make = make
        self.model = model
        self.year = year
        self.sold_on = sold_on
        """""""

    def sale_price(self):
        """""""Return the sale price for this vehicle as a float amount."""
        if self.sold_on is None:
            return 0.0  # Already sold
        return 5000.0 * self.wheels

    def purchase_price(self):
        """""""Return the price for which we would pay to purchase the vehicle."""
        if self.sold_on is None:
            return 0.0  # Not yet sold
        return self.base_sale_price - (0.10 * self.miles)

@abstractmethod
def vehicle_type(self):
    """""""Return a string representing the type of vehicle this is."""
    pass
Inheritance once again

class Car(Vehicle):
    """A car for sale by Jeffco Car Dealership."""
    base_sale_price = 8000
    wheels = 4

    def vehicle_type(self):
        """Return a string representing the type of vehicle this is.""
        return 'car'

class Truck(Vehicle):
    """A truck for sale by Jeffco Car Dealership."""
    base_sale_price = 10000
    wheels = 4

    def vehicle_type(self):
        """Return a string representing the type of vehicle this is.""
        return 'truck'
Demo