ECE 20875

Python for Data Science

Chris Brinton, Qiang Qiu, and Mahsa Ghasemi

(Adapted from material developed by Profs. Milind Kulkarni, Stanley Chan, Chris Brinton, David Inouye, Qiang Qiu)

inheritance

reusing functionality

- We often want to reuse functionality from an existing class
  - When a new class that has some extra functionality compared to an old class
  - When a new class changes/overrides some functionality of an old class
- One option: Create a new class, and define all the necessary functions
  - This is done in the example on the right

```
class Person :
  def __init__(self, name) :
    self.name = name

  def getName(self) :
    return self.name

p = Person(“Bob”)
print(p.getName()) #prints “Bob”

#creating a new class that has a lot in
common with “Person”
class AgePerson :
  def __init__(self, name, age) :
    self.name = name
    self.age = age

  def getName(self) :
    return self.name

  def getAge(self) :
    return self.age
```

inheriting from parent class

- This is pretty inefficient if there is a lot of overlap
- Instead, we can use **inheritance**
  - Create a new **child** class that *inherits* the attributes of the **parent** class
  - Can then add new attributes to a class to define new functions and/or add new data
- Updated example using inheritance on the right:
  - `__init__` from `AgePerson` **overrides** `__init__` from `Person`
  - When we create a new `AgePerson`, we use the new version of `__init__`, but when we call `getName()`, we use the old version of `getName()`

```
class Person :
  def __init__(self, name) :
    self.name = name

  def getName(self) :
    return self.name

p = Person("Bob")
print(p.getName())

#we can instead let the AgePerson class
inherit from the parent class Person
class AgePerson(Person) :
  #overrides __init__ from parent
  def __init__(self, name, age) :
    self.name = name
    self.age = age

  def getAge(self) :
    return self.age
```

reusing when redefining

- Can reuse functionality even more by using the **`super()`** function within a child class
  - Tells the class to inherit this method/property from the parent, and allows further redefining
- Updated example on the right:
  - `super().__init__()` refers to `__init__()` of the parent class `Person`
  - This tells `AgePerson` to reuse `__init__` from `Person` in the redefinition, and then we can add additional functionality on top of it
- Can similarly reuse functionality when redefining other functions

```
class Person :
  def __init__(self, name) :
    self.name = name

  def getName(self) :
    return self.name

p = Person("Bob")
print(p.getName())

class AgePerson(Person) :
  def __init__(self, name, age) :
    #Tell AgePerson to inherit __init__
    from parent class
    super().__init__(name)

    #Then we can add additional
    functionality to the new init
    self.age = age

  def getAge(self) :
    return self.age
```

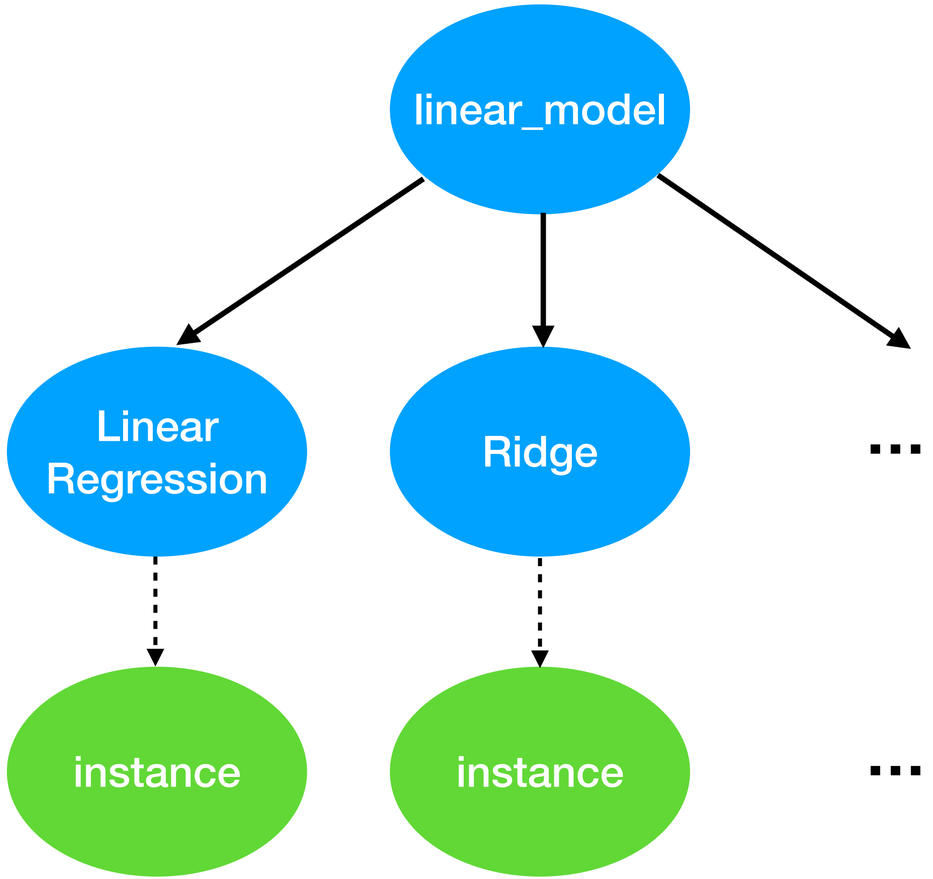
overriding default methods

- All classes inherit from the built-in basic class called **`object`** by default
  - Provides some default functionality like `__str__` and **`__repr__`** methods
  - **__repr__** is the “official” string representation of an object, more general than just printing, useful for debugging
  - `__str__` is the “informal” string representation of an object, used for creating readable end user output
- Overriding these gives us the ability to change how objects are represented (`__repr__`) or printed (`__str__` or `__repr__`)

```
class Person :
  def __init__(self, name) :
    self.name = name

  def getName(self) :
    return self.name

p = Person(“Bob”)
print(p.getName())

class AgePerson(Person) :
  def __init__(self, name, age) :
    super().__init__(name)
    self.age = age

  def getAge(self) :
    return self.age

  def __repr__(self) :
    return self.name + “, “ + str(self.age)

p = AgePerson(“Bob”, 33)
repr(p)   #prints ‘Bob, 33’
```

uses of inheritance we've seen

- We've seen inheritance used in many Python packages we have used in this class
- Distribution classes (normal, exponential, etc.) in `sklearn` all inherit from generic classes that provide some default functionality
  - These classes override key methods (like `pdf` and `cdf`) to provide distribution-specific implementations
- Several regression models in `sklearn` inherit functionality from `linear_model`



what about polymorphism or interfaces?

- You may have heard of **polymorphism** before
  - Call a function on an object, but invoke different functionality depending on exactly what class an object is
  - Can write very generic code since you do not have to know exactly what type of object you are working with
  - Used extensively in languages like Java and C++ through the inheritance mechanism
- Python gets you this “for free”:
  - Programs are not written with types
  - Invoke any method on any object if the object’s class has the method defined (called **duck typing**)
  - No need for any actual relationship between different classes that implement the same method(s)

```
class Animal :
  def __init__(self, name) :
    self.name = name

  def talk(self) :
    raise NotImplementedError(“Subclass
            must implement talk method”)

class Cat(Animal) :
  def talk(self):
     return ‘Meow!’

class Duck:  # Notice doesn’t inherit
  def __init__(self, name) :
     self.name = name # But has the right var.
  def talk(self): # And implements this method
     return ‘Quack! Quack!’

animals = [Cat(‘Missy’), Cat(‘Mr. Mistoffelees’),
                    Duck(‘Sammy’)]

for animal in animals:
        print(animal.name + ‘: ‘ + animal.talk())
```