ECE 20875
Python for Data Science
Milind Kulkarni and Chris Brinton

inheritance
reusing functionality

• We often want to reuse functionality from an existing class

• A new class that has some extra functionality compared to an old class

• Or that changes/overrides some functionality of an old class

• One option: create a new class and define all the necessary functions

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

    def getName(self):
        return self.name

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

class OldPerson:
    def __init__(self, name, age):
        self.name = name
        self.age = age

    def getName(self):
        return self.name

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

• Instead we can use inheritance
  • Create a new class that inherits the attributes of the old class
  • Can then add new attributes to a class to define new functions, add new data
  • Note that __init__ was overridden

• When we create a new OldPerson, we use the new version of __init__ but when we call getName() we use the old version of getName()

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

    def getName(self) :
        return self.name

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

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

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

• Can reuse functionality even more by calling “super class” functions

• No need to copy code from one __init__ function to another

• Can similarly reuse functionality when redefining other functions

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

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

class OldPerson(Person):
    def __init__(self, name, age):
        super().__init__(self, name)
        self.age = age
    def getAge(self):
        return self.age
```
method overriding

• All classes inherit from **object** by default
  
• Provides some default functionality like **__repr__** and **__str__** methods
  
• Overriding these gives us the ability to change how objects are represented or printed

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

    def getName(self):
        return self.name

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

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

    def getAge(self):
        return self.age

    def __repr__(self):
        return name + ", " + str(self.age)
```
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 scipy 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 LinearModel
what about polymorphism?

• 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
    
    • No need for any actual relationship between different classes that implement the same method(s)