CSCI 134 Fall 2021:
Classes, Objects, and Inheritance

Nov 8, 2021

Shikha Singh, 9AM
Jeannie Albrecht, 10AM
Announcements & Logistics

- **Lab 8 (Ciphers)** is a partner lab: focuses on using classes and (a little) inheritance
  - Must attend one lab session together
  - Mon lab due on Wed, Tue lab due on Thur
- **Lab 6** feedback returned on Friday
- **HW 7** due tonight
- Thanksgiving week
  - We’ll have lecture on Monday 11/22 as usual
  - Lab on Mon/Tue afternoon will be optional but encouraged if you’re on campus

Do You Have Any Questions?
Last Time

• Built the **Coordinate class** to represents points on a plane
• Learned about private and public attributes (signal using underscores)
• Explored about getter and setter methods in Python (and special `@` annotations)
• Discussed bigger OOP ideas:
  • Abstraction and data hiding
  • Encapsulation
Today’s Plan

• Learn about more special __ (double underscore) methods
  
  • __str__ : print representation of objects
  
  • __repr__ : string representation of objects

• Begin talking about inheritance
Special methods and attributes

• We’ve seen several “special” methods and attributes in Python:
  • __all__ list of names in module to be imported by *
  • __name__ special module attribute
  • __main__ name attribute of scripts
  • __init__ method
  • __slots__ list for static attributes
  • __dict__ list for dynamic attributes
• Today we’ll look at two more special methods:
  • __str__ and __repr__
Special method `__str__` is called when we print a class object.

We can customize how the object is printed by writing a `__str__` method for our class.

`__str__` always returns a string.

We can choose how the objects of our class are printed!
Defining the \_str\_ method

class Coordinate(object):
    \_slots__ = ['\_x', '\_y']
    def \_init\_(self, x, y):
        self.\_x = x
        self.\_y = y

    # other methods
    def \_str\_(self):
        return "<{{}, {{}}>".format(self.\_x, self.\_y)

>>> print(pt)
<3, 4>
String Representation of an Object

- Special method `__repr__` is used to create a string representation of an object.
- `__repr__` is called when we call "eval" on an object (`eval` is called in interactive python and we ask for the value of an object).

```python
In [1]: class A:
   """Testing repr method""
   pass

In [2]: a = A()

In [3]: a

Out[3]: <__main__.A at 0x111dcf8b0>
```

By default, if we eval an object, its not “pretty”
String Representation of an Object

- Special method **`__repr__`** is used to create a string representation of an object.

- **`__repr__`** is called when we call "**`eval`**" on an object (**`eval`** is called in interactive python and we ask for the value of an object).

```
In [4]: class A:
    """Testing repr method""
    
def __repr__(self):
        return "A()"

In [5]: a = A()

In [6]: a
Out[6]: A()

In [7]: type(a)
Out[7]: __main__.A
```

Notice we didn’t say “print” so **`__repr__`** is called rather than **`__str__`**.
Defining the `__repr__` method

- Special method `__repr__` is used to create a string representation of an object.
- `__repr__` is called when we call "eval" on an object (eval is called in interactive python and we ask for the value of an object).
- `__repr__` always returns a string.

```python
def __repr__(self):
    return "Coordinate({},{}).format(self.x, self.y)
```

```python
pt = Coordinate(3, 4)

pt  # calls repr

Coordinate(3,4)
```
Name Class
Example: Name Class

- Names of people have certain attributes
  - Almost everyone has a **first and last name**
  - Some people have a **middle name**
- We can create name objects by defining a class to represent these attributes
- Then we can define methods, e.g., getting initials of people's names, etc
- Let's practice some of the concepts using this class
  - ```str```: how do we want the names to be printed?
  - ```repr``` : what do we want name objects to look like in interactive python?
  - **initials**: can we define a property that returns the initials of people's names?
Example: Name Class

```
In [13]:
class Name:
    """Class to represent a person's name."""
    __slots__ = ['_f', '_m', '_l']

    def __init__(self, first, last, middle=''):  
        self._f = first
        self._m = middle
        self._l = last

    def __str__(self):
        if len(self._m):  # if the person has a middle name
            return '{}. {}{}{}'.format(self._f[0], self._m[0], self._l)
        return '{}{}{}'.format(self._f[0], self._l)

In [14]:
n1 = Name('Shikha', 'Singh')
n2 = Name('Jeannie', 'Albrecht', 'R. ')

In [15]:
print(n1)
print(n2)

S. Singh
J. R. Albrecht
```
Example: Name Class

```python
In [37]: class Name:
    
    """Class to represent a person's name."""
    __slots__ = ['_f', '_m', '_l']

    def __init__(self, first, last, middle=' '):
        self._f = first
        self._m = middle
        self._l = last

    @property
    def initials(self):
        if len(self._m):
            return '{}, {}, {}'.format(self._f[0], self._m[0], self._l[0]).upper()
        return '{}, {}, {}'.format(self._f[0], self._l[0]).upper()

    def __str__(self):
        if len(self._m):  # if the person has a middle name
            return '{}, {}, {}'.format(self._f[0], self._m[0], self._l)
        return '{}, {}'.format(self._f[0], self._l)

    def __repr__(self):
        if len(self._m):
            return "Name('{}', '{}', '{}')".format(self._f, self._l, self._m)
        return "Name('{}', '{}')".format(self._f, self._l)

In [38]: n = Name('Ruth', 'Bader', 'Ginsburg')

In [39]: n.initials  # notice no parenthesis!

Out[39]: 'R. G. B.'
```
Other Special Methods

• There are many other special methods that we won’t discuss in detail. Suppose we have two instances x and y:
  • __len__(self): len(x)
  • __contains__(self, item): item in x
  • __eq__ (self, other): x == y
  • __lt__ (self, other): x < y
  • __gt__ (self, other): x > y
  • __add__ (self, other): x + y
  • __sub__ (self, other): x - y
  • __mul__ (self, other): x * y
  • __truediv__ (self, other): x / y
  • __pow__ (self, other): x ** y
• There are others!

We’ll come back to these when we discuss linked lists.
Inheritance
Introduction to Inheritance

• **Inheritance** is the capability of one class to derive or *inherit* the properties from another class

• The benefits of inheritance are:
  • Often represents real-world relationships well
  • Provides **reusability of code**, so we don’t have to write the same code again and again
  • Allows us to add more features to a class without modifying it

• Inheritance is **transitive** in nature, which means that if class B inherits from class A, then all the subclasses of B would also automatically inherit from class A

• When a class inherits from another class, all methods and attributes are accessible to subclass, *except private attributes* (indicated with __)
Inheritance Example

• Suppose we have a base class Fish

• Fish defines several methods that are common to all fish:
  • eat(), swim()

• Fish also defines several attributes with default values:
  • _length, _weight, _lifespan
Inheritance Example

• All fish have some features in common
  • But not all fish are the same!

• Each Fish instance will specify different values for attributes (_length, _weight, _lifespan)

• Some fish may still need extra functionality!
Inheritance Example

• For example, Sharks might need an `attack()` method
• Pufferfish might need a `puff()` method
• We might even want to `override` an existing method with a different (more specialized) implementation
  • Inheritance allows for all of this!
Inheritance

• When defining super/parent classes, think about the common features and methods that all subclasses will have

• In subclasses, inherit as much as possible from super class, and add and/or override attributes and methods as necessary

• Consider an example:
  
  • **Person** class: defines common attributes for all people on campus
  
  • **Student** subclass: inherits from **Person** and adds additional attributes for student’s **major** and **year**
  
  • **Faculty** subclass: inherits from **Person** and adds additional attributes for **department** and **office**
  
  • **Staff** subclass: inherits from **Person** and adds additional attributes for type/status of employee (**full-time, part-time**)
class Person:
    __slots__ = ['_name']

    def __init__(self, name):
        self._name = name

    def __str__(self):
        return self._name

    def __repr__(self):
        return "Person('{}')".format(str(self._name))

@property
def name(self):
    return self._name
Student Class

```python
class Student(Person):
    __slots__ = ['_year', '_major']

    def __init__(self, name, year, major):
        super().__init__(name)  # call __init__ of super class
        self._year = year
        self._major = major

@property
def year(self):
    return self._year

@property
def major(self):
    return self._major

@major.setter
def major(self, major):
    self._major = major
```

Notice this does not include the inherited attribute '_name'

This calls the __init__ method of Person
Using the Student Class

In [21]: jane = Student("Jane", 2024, "CS")

In [22]: jane.name  # inherited from Person
Out[22]: 'Jane'

In [23]: jane.major  # defined in Student
Out[23]: 'CS'

In [24]: jane.major = "Math"

In [25]: jane.major
Out[25]: 'Math'
class Faculty(Person):
    __slots__ = ['_dept', '_office']

    def __init__(self, name, dept, office):
        super().__init__(name)  # call __init__ of super class
        self._dept = dept
        self._office = office

@property
def dept(self):
    return self._dept

@property
def office(self):
    return self._office
Using the Faculty Class

```python
In [29]: shikha = Faculty("Shikha", "CS", "TCL 304")

In [30]: shikha.name
Out[30]: 'Shikha'

In [31]: shikha.dept
Out[31]: 'CS'

In [33]: shikha.major

---------------------------------------------------------------------------
AttributeError                       Traceback (most recent call last)
<ipython-input-33-4ee0b7f198a3> in <module>
----> 1 shikha.major

AttributeError: 'Faculty' object has no attribute 'major'
```
Using the Faculty Class

In [29]: shikha = Faculty("Shikha", "CS", "TCL 304")

In [30]: shikha.name
Out[30]: 'Shikha'

In [31]: shikha.dept
Out[31]: 'CS'

In [33]: shikha.major

---

```
AttributeError
Traceback (most recent call last)
<ipython-input-33-4ee0b7f198a3> in <module>
----> 1 shikha.major

AttributeError: 'Faculty' object has no attribute 'major'
```
Staff Class

class Staff(Person):
    __slots__ = ['_fulltime']

def __init__(self, name, fulltime):
    # fulltime is a Boolean
    super().__init__(name)  # call __init__ of super class
    self._fulltime = fulltime

@property
def status(self):
    if self._fulltime:
        return "fulltime"
    return "parttime"

Notice that @property methods can do more than just return an attribute directly
Using the Staff Class

In [37]:
fred = Staff("Fred", False)

In [38]:
print(fred)

Fred

This calls __str__ of the Person class

In [40]:
fred.status

Out[40]: 'parttime'
Summary

• Inheritance is a very useful feature of OOP
• Supports code reusability
• One superclass can be used for the number of subclasses in a hierarchy
• Can change the parent class without changing the subclasses
• More next time!