CSCI 134 Fall 2021:
Classes and Objects II

Nov 05, 2021

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

- **Lab 8 (Ciphers)** is going to be a **partner lab**
  - Google form (from Lida) was due **yesterday @ 10 pm**
  - **Both partners have to fill out the form!**
  - **Must attend one lab session together**
  - Mon lab due on Wed, Tue lab due on Thur
  - Can work by yourself but **strongly encourage** you to find a partner

- **Lab 6** graded feedback: coming soon (sorry for the delay)
- **HW 7** due Mon 10 pm (fewer questions this week)
- **Class of 60s Colloquium today** 2:35 @ Wege (TCL 123)

**Do You Have Any Questions?**
Last Time

• Introduced the big idea of **object oriented programming**
  • Everything in Python is an object and has a type!
    • We can create classes to define our own types
• Defined a simple Book class and learned about the following:
  • data attributes of objects, methods to interact with objects
  • `class` keyword to define a class
  • `self` parameter in methods of a class that receives a reference to the calling instance
  • defining and calling methods on objects of a class
  • `__init__` method to initialize objects with their attributes
  • `__slots__` list to fix the attributes an object of a class can have
class Book:
    """This class represents a book with attributes name, author and year""
    __slots__ = ['name', 'author','year']  # attributes

def __init__(self, name=\_\_\_None\_\_\_, author=\_\_\_None\_\_\_, year=\_\_\_None\_\_\_):
    self.name = name
    self.author = author
    self.year = year

def numWordsName(self):
    """Returns the number of words in name of book""
    return len(self.name.split())

def sameAuthorAs(self, other):
    """Check if self and other have same author""
    return self.author == other.author

def yearsSincePub(self, currentYear):
    """Returns the number of years since book was published""
    return currentYear - self.year
Defining More Methods

• We can define the following methods in the class definition of Book to provide an interface to our book objects:
  
  • **numWordsName**: returns the number of words in the name of the book
  • **sameAuthorAs**: takes another book object as a parameter and checks if the two books have the same author or not
  • **yearSincePub**: takes in the current year and returns the number of years since the book was published
class Book:
    """This class represents a book with attributes name, author and year""

    __slots__ = ['name', 'author', 'year']  # attributes

    def __init__(self, name=None, author=None, year=None):
        self.name = name
        self.author = author
        self.year = year

    def numWordsName(self):
        """Returns the number of words in name of book""
        return len(self.name.split())

    def sameAuthorAs(self, other):
        """Check if self and other have same author""
        return self.author == other.author

    def yearsSincePub(self, currentYear):
        """Returns the number of years since book was published""
        return currentYear - self.year
Review: Using the Book Class

```python
# creating book objects:
p = Book('Pride and Prejudice', 'Jane Austen', 1813)
em = Book('Emma', 'Jane Austen', 1815)
```

```python
In [3]: pp.sameAuthorAs(em)
Out[3]: True
```

```python
In [4]: pp.sameAuthorAs(hp)
Out[4]: False
```

```python
In [5]: hp.numWordsName()
Out[5]: 6
```

```python
In [6]: emma.yearsSincePub(2020)
Out[6]: 205
```

```python
In [7]: hp.yearsSincePub(2020)
Out[7]: 23
```
Today’s Plan

• Build the **Coordinate class** to represents points on a plane

• Learn about how Python handles private and public attributes
  • Use underscores to signal private vs public attributes

• Explore getter and setters in Python

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

• Discuss bigger OOP ideas:
  • Abstraction and data hiding
  • Encapsulation
Data Abstraction

• Python supports **data abstraction** (separating the data and details of the implementation from the user) via:

  • **Data hiding**: via attribute naming conventions (private, public)
  • **Encapsulation**: bundling together related data and methods that provide an interface to the data
Attribute Naming Conventions

• Double leading underscore (___) in name (strictly private): e.g. __val
  • Invisible from outside of the class
  • Strong “you cannot touch this” policy
• Single leading underscore (_ ) in name (private/protected): e.g. _val
  • Can be accessed from outside, but really shouldn’t
  • “Don’t touch this unless you are a subclass” policy
• No leading underscore (public): e.g. val
  • Can be freely used outside class
• Conventions apply to procedural attributes (methods names) as well!
• Note: Like many things in Python, some of these are conventions, not rules!
Attribute Naming Conventions

```python
In [1]: class TestingAttributes:
   ...:     __slots__ = ['__val', '_val', 'val']
   ...:     def __init__(self):
   ...:         self.__val = "I am strictly private."
   ...:         self._val = "I am private but accessible from outside."
   ...:         self.val = "I am public."

In [2]: a = TestingAttributes()

In [3]: a.__val

-------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-3-3e19e2bd1a2b> in <module>
----> 1 a.__val

AttributeError: 'TestingAttributes' object has no attribute '__val'

In [4]: a.val

Out[4]: 'I am private but accessible from outside.'

In [5]: a.val

Out[5]: 'I am public.'
```
Methods and Data Abstraction

• Methods of a class typically fall into two categories:
  
  • **accessor (getter) methods**: provide “read-only” access to the object’s attributes
  
  • **mutator (setter) methods**: let us modify the object’s attribute values

• Ideally, we do not allow the user direct access to the object’s attributes

• Instead we **control access** through accessor and mutator methods

• This approach enforces **data abstraction**
  
  • Methods provide a public interface to attribute values
  
  • Attributes are part of the private implementation
Coordinate Class

(-5,2)  ○
(-5,-2)  ○
(2,5)  ○
(5,2)  ○
(5,-2)  ○

y-axis

y = x

x-axis
Let us write a class `Coordinate` which represents a point \((x, y)\).

Notice that this class definition has \(\text{object}\) after the class name just like a function definition.

- Here \texttt{object} refers to the \texttt{parent} class of the class \texttt{Coordinate}.

We will talk more about super/parent classes and subclasses when we discuss Inheritance in Python (next week).

```python
In [1]: class Coordinate(object):
   ...:     """Represents the coordinates of a point."""
   ...:     def __init__(self, x, y):
   ...:         self.x = x
   ...:         self.y = y
```
Coordinate Class

• Reminder: we use the `class` keyword to define a new type

```python
class Coordinate(object):
    # define attributes here
    # indented body of class definition
```

• `object` means `Coordinate` is a Python `object` and inherits all of its attributes (inheritance will be covered next week)

• `Coordinate` is a subclass of `object`

• `object` is a superclass of `Coordinate`
Initializing the Class: `__init__`

- Recall `__init__` lets us initialize some data attributes of the class.
- Recall `__slots__` stores the data attribute names as strings in a list.
- Single leading underscore signals private data or procedural attribute.

```python
class Coordinate(object):
    __slots__ = ['_x', '_y']

def __init__(self, x, y):
    self._x = x
    self._y = y
```

- Parameter to refer to an instance of the class.
- Can assign values to an instance of a class using dot notation.

Single leading underscore: private data attributes.
OOP Principle: Encapsulation

- **Encapsulation** is the bundling of data with the methods that operate on that data.

- It is often accomplished by providing two kinds of procedural attributes:
  - **getter or accessor methods:** methods for retrieving or accessing the values of attributes
    - Getter methods do not change the values of attributes, they just return the values, and
  - **setter or mutator methods:** methods used for changing the values of attributes
    - They do not return anything

- These are not specific to Python but exist in every OOP language.
Accessor Methods

- Each Coordinate has two private data attributes \_x, and \_y
- We can write accessor methods to provide access to these attributes

```python
class Coordinate(object):
    """Represents the coordinates of a point."""
    __slots__ = [\'_x', \'_y']

    def __init__(self, x, y):
        self._x = x
        self._y = y

    def getX(self):
        return self._x

    def getY(self):
        return self._y
```
Accessor Methods

- This is how accessors methods are written in other languages
- Python also provides a way to define methods as if they were data attributes (without the () notation) through @property annotation

```python
class Coordinate(object):
    """Represents the coordinates of a point."""
    __slots__ = ["_x", "_y"]

    def __init__(self, x, y):
        self._x = x
        self._y = y

    def getX(self):
        return self._x

    def getY(self):
        return self._y
```
**Accessor Methods via @property**

- **Annotations @.** Python provides a rich collection of syntactic notes that can change how code is interpreted, called annotations.

- These are typically prefixed with the at-sign (@).

- Accessor methods **do not change the state of the calling object** and are used just to retrieve some information about the object.

- **@property annotation.** Treat a procedural attribute as a data attribute:
  - If we’d like to treat an accessor method as if it were a data attribute, we can use the @property annotation.
Accessor Methods via `@property`

```python
class Coordinate(object):
    """Represents the coordinates of a point."""
    slots = ['_x', '_y']
def __init__(self, x, y):
    self._x = x
    self._y = y

@property
def x(self):
    return self._x

@property
def y(self):
    return self._y

In [9]: p = Coordinate(4, 3)
q = Coordinate(0, 2)

In [10]: p.x  # notice no parenthesis!
Out[10]: 4
```
Accessor Methods via @property

```python
class Coordinate(object):
    """Represents the coordinates of a point."""
    slots = ["_x", "_y"]
def __init__(self, x, y):
    self._x = x
    self._y = y

@property
def x(self):
    return self._x

@property
def y(self):
    return self._y
```

```python
In [12]: p.x + q.x, p.y + q.y
Out[12]: (4, 5)
```
Setter Method \texttt{f} via \texttt{@f.setter}

- Sometimes we need to define methods that mutate the property of an object (say update the \(x\) and \(y\) coordinate of a point)

- These methods are called \textbf{setter methods}

- Like getter methods, we can define setter methods for properties using the \texttt{@} annotation

- This has the advantage of allowing you to write

\[
\text{pt} = \text{Coordinate}(0, 0)
\]

\[
\text{pt.x} = 10
\]

Instead of

\[
\text{pt.setX}(10)
\]
class Coordinate(object):
    """Represents the coordinates of a point."""
    __slots__ = ['_x', '_y']
def __init__(self, x, y):
    self._x = x
    self._y = y

@property
def x(self):
    return self._x

@property
def y(self):
    return self._y

@x.setter
def x(self, newX):
    self._x = newX

@y.setter
def y(self, newY):
    self._y = newY
Setter Method $f$ via `@f.setter`

- Cannot define `@f.setter` if property $f$ is not defined
- Setters and properties go hand in hand

```python
class Test:
    
def __init__(self, a):
        self.a = a

    @f.setter
    def f(self, new_val):
        self.a = new_val
```

```
NameError                               Traceback (most recent call last)
/var/folders/h8/n5myy3jdld7c6v42cw42flt80000gn/T/ipykernel_96527/1536826767.py in <module>
    4        self.a = a
    5
/var/folders/h8/n5myy3jdld7c6v42cw42flt80000gn/T/ipykernel_96527/1536826767.py in Test()
    4        self.a = a
    5
----> 6        @f.setter
    7        def f(self, new_val):
    8        self.a = new_val

NameError: name 'f' is not defined
```
Coordinate Class

- What methods are useful when dealing with coordinate objects?
  - One might want to find the distance between coordinates
- Euclidean distance is defined as follows

\[ d = \sqrt{(x_2-x_1)^2 + (y_2-y_1)^2} \]
Other Methods: See Notebook
For next time:

```python
__str__ & __repr__
```
Print Representation of an Object

• 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

• We can choose how the objects of the class are printed!

```python
In [1]: class A:
   ...:     """Test printing of objects.""
   ...:     pass

In [2]: a = A()

In [3]: print(a)
   ...: <__main__.A object at 0x111e90750>
```

By default, if we print an object, it's not “pretty”
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 a string (`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>
```
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 a string (eval is called in interactive python and we ask for the value of an object)

```python
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
```
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 a string (eval is called in interactive python and we ask for the value of an object)

```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 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

```python
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.'
```
Acknowledgments

These slides have been adapted from:

- https://www.python-course.eu/python3_object_oriented_programming.php