CS 134:
Classes and Objects (2)

Announcements & Logistics

- **Lab 7** due today/tomorrow
- **Lab 8** is going to be a **partner lab**
  - Look for a Google form from Lida
  - **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 encouraged** to find a partner
- **Lab 6** graded feedback: coming soon (sorry for the delay)
- **HW 7** due Mon 11 pm (fewer questions this week)
- **CS info session this Friday** (learn about major requirements and courses being offered next year): 2:35 @ Wege (TCL 123)

Do You Have Any Questions?

Last Time

- Introduced the big idea of **object oriented programming** (OOP)
- Everything in Python is an object and has a type!
    - We can create **classes** to define our own types
- Learned about using the `class` keyword to define a class
- Reviewed how to define and call **methods** on objects of a class
    - Methods facilitate **abstraction:** hide unnecessary implementation details
    - Discussed using the `self` parameter in methods of a class (`self` is a a reference to the calling instance)
- Quick aside: **functions versus methods**?
    - Functions are not associated with a specific class
    - Methods are associated with a specific class and are invoked on instances of the class (using dot notation)

Today's Plan

- Implement a simple Book class and learn about the following:
  - Declaring data **attributes** of objects using `__slots__`
  - Learning about scope and naming conventions in Python
  - Using the `__init__()` method to initialize objects with their attribute values
  - Defining accessor and mutator methods to interact with attributes
  - Implementing and invoking methods in general
  - Implementing `__str__()` method to provide meaningful print statements for custom objects

Defining a Class

- Key features of a class:
  - **Attributes** that describe instance-specific data
  - **Methods** that act on those attributes
- When defining a new class (aka an object blueprint), it's important to identify what attributes are required and what actions will be performed using those attributes
- For example, suppose we want to define a new Book class
  - Attributes?
  - Methods?

Defining a Class

- Key features of a class:
  - **Attributes** that describe instance-specific data
  - **Methods** that act on those attributes
- When defining a new class (aka an object blueprint), it's important to identify what attributes are required and what actions will be performed using those attributes
- For example, suppose we want to define a new Book class
  - Attributes?
    - Title, author, publication year, genre, …
  - Methods?
    - sameAuthorAs( ), yearsSincePub( ), …

Defining Our Own Class: Book

Name of class: Capitalized by convention

```
class Book:
    """This class represents a book"""
    # attributes go here
    # indented body of class definition (methods, etc)
```

Creating instances of the class:

```
b1 = Book()
```

b1 is an instance of class Book

```
b2 = Book()
```

b2 is another (different) instance of class Book

Attributes

- Objects have *state* which is typically held in **instance variables** or (in Pythonic terms) **attributes**.
- Example: For our **Book** class, these include the book's title, author, and publication year
- Every **Book** instance has different attribute *values*!
- In Python, we **declare** attributes using `__slots__`
- `__slots__` is a **list of strings** that stores the **names** of all attributes in our class (note that only names of attributes are stored, not the values!)
- `__slots__` is typically defined at the top of our class (before method definitions)

Declaring Attributes in `__slots__`

```
class Book:
    """This class represents a book"""
   # declare Book attributes
    __slots__ = [“author”, “title”, “year”]


    # indented body of class definition
```

“author”, “title”, and “year” are **attributes** of the Book class

Scope and Naming Conventions in Python

- Double leading underscore (__) in attribute name (**strictly private**): e.g. `__value`
  - "Invisible" from outside of the class
  - Strong ***"you cannot touch this"*** policy
- Single leading underscore (_) in name (**private/protected**): e.g. `_value`
  - 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. `value`
  - Can be freely used outside class
- Conventions apply to **methods names** as well!
- Note: In Python, these are conventions, not rules! But we'll follow them

Attribute Naming Conventions

```
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.'
```

Note: Just because we *can* access attributes directly using dot notation, doesn't mean we *should*! We'll come back to this…

Declaring Attributes in `__slots__`

```
class Book:
    """This class represents a book"""
   # declare Book attributes
    __slots__ = [“_author”, “_title”, “_year”]

    # indented body of class definition
```

“_author”, “_title”, and “_year” are protected **attributes** of the Book class

Initializing a Class: `__init__`

- How do we assign values to the attributes in `__slots__`?
- Attributes should be assigned initial values as part of the class definition
- We can achieve this using the `__init__` method in Python
  - Like a constructor in Java (more on this in a few weeks)
- The `__init__` method is **run anytime a new instance of a class is created**

```
In [1]: class TestInit:
            """This class will test when __init__ is called"""
            def __init__(self):
                print("__init__ is called")
```

```
In [2]: obj = TestInit()
```

```
__init__ is called
```

Book class: `__init__`

- In most cases, the `__init__` method should set values for the class attributes declared in slots
- Values are often provided as parameters to `__init__`

```
class Book:
    """This class represents a book with attributes title, author, and year"""

    # attributes
    # _ indicate that they are protected
    __slots__ = ['_title', '_author', '_year']

    def __init__(self, bookTitle, bookAuthor, bookYear):
        self._title = bookTitle
        self._author = bookAuthor
        self._year = bookYear
```

When referring to class attributes, use self.{attribute name}.

```
In [3]: # creating book objects:
        pp = Book('Pride and Prejudice', 'Jane Austen', 1813)
        emma = Book('Emma', 'Jane Austen', 1815)
        hp = Book("Harry Potter and the Sorcerer's Stone", "J.K. Rowling", 1997)

In [5]: hp._title

Out[5]: "Harry Potter and the Sorcerer's Stone"
```

An Aside: Default Argument Values

- Python supports the ability to provide default argument values in method and function definitions

```
class Book2:
    """This class represents a book with attributes title, author, and year"""

    # attributes
    __slots__ = ['_title', '_author', '_year']

    # this __init__ method specifies default values for the parameters
    def __init__(self, bookTitle="", bookAuthor="", bookYear=0):
        self._title = bookTitle
        self._author = bookAuthor
        self._year = bookYear
```

- If we create a Book and don't provide values for the arguments in `__init__`, the values are set to be the default values ("" and 0 in this case)

```
In [7]: emptyBook = Book2()

In [8]: emptyBook._title
Out[8]: ''
```

- For now, we'll remove these default values for simplicity

Methods and Data Abstraction

- Ideally, we should not allow the user direct access to the object's attributes:

```
In [9]: # creating book objects:
        hp = Book("Harry Potter and the Sorcerer's Stone", "J.K. Rowling", 1997)

In [10]: hp._title
Out[10]: "Harry Potter and the Sorcerer's Stone"
```

- Instead we control access to attributes through accessor and mutator methods and avoid accessing the attributes directly
    - **Accessor methods:** provide "read-only" access to the object's attributes ("getter" methods)
    - **Mutator methods:** let us modify the object's attribute values ("setter" methods)
- This is called **encapsulation**: the bundling of data with the methods that operate on that data (another OOP principle)

```
class Book:
    """This class represents a book with attributes title, author, and year"""

    # attributes
    # _ indicate that they are protected
    __slots__ = ['_title', '_author', '_year']

    def __init__(self, bookTitle, bookAuthor, bookYear):
        self._title = bookTitle
        self._author = bookAuthor
        self._year = bookYear

    def getTitle(self):
        return self._title

    def getAuthor(self):
        return self._author

    def getYear(self):
        return self._year

    def setTitle(self, bookTitle):
        self._title = bookTitle

    def setAuthor(self, bookAuthor):
        self._author = bookAuthor

    def setYear(self, bookYear):
        self._year = int(bookYear)
```

Accessor methods return values of attributes, but do not change them

```
class Book:
    """This class represents a book with attributes title, author, and year"""

    # attributes
    # _ indicate that they are protected
    __slots__ = ['_title', '_author', '_year']

    def __init__(self, bookTitle, bookAuthor, bookYear):
        self._title = bookTitle
        self._author = bookAuthor
        self._year = bookYear

    def getTitle(self):
        return self._title

    def getAuthor(self):
        return self._author

    def getYear(self):
        return self._year

    def setTitle(self, bookTitle):
        self._title = bookTitle

    def setAuthor(self, bookAuthor):
        self._author = bookAuthor

    def setYear(self, bookYear):
        self._year = int(bookYear)
```

Mutator methods change the value of attributes but do not explicitly return anything

Using Accessor/Mutator Methods

```
In [5]: pp.getTitle()
Out[5]: 'Pride and Prejudice'

In [6]: emma.getAuthor()
Out[6]: 'Jane Austen'

In [10]: hp.getYear()
Out[10]: 1997

In [11]: hp.setYear(1998)

In [12]: hp.getYear()
Out[12]: 1998
```

Use accessor methods to get the values of the attributes (when outside of class implementation)

Use mutator methods to set or change the values of the attributes (when outside of class implementation)

Defining More Methods

- Beyond the accessor and mutator methods, we can define other methods in the class definition of `Book` to manipulate or answer questions about our book objects:
    - `numWordsInTitle():` returns the number of words in the title of the book
    - `yearSincePub(currentYear):` takes in the current year and returns the number of years since the book was published
    - `sameAuthorAs(otherBook):` takes another Book object as a parameter and checks if the two books have the same author or not

```python
class Book:
    """This class represents a book with attributes title, author, and year"""

    # attributes
    # _ indicates that they are protected
    __slots__ = ['_title', '_author', '_year']

    # __init__ is automatically called when we create new Book objects
    # we set the intial values of our attributes in __init__
    def __init__(self, bookTitle, bookAuthor, bookYear):
        self._title = bookTitle
        self._author = bookAuthor
        self._year = bookYear

    # accessor (getter) methods
    def getTitle(self):
        return self._title

    def getAuthor(self):
        return self._author

    def getYear(self):
        return self._year

    # mutator (setter) methods
    def setTitle(self, bookTitle):
        self._title = bookTitle

    def setAuthor(self, bookAuthor):
        self._author = bookAuthor

    def setYear(self, bookYear):
        self._year = int(bookYear)

    # methods for manipulating Books
    def numWordsInTitle(self):
        """Returns the number of words in name of book"""
        return len(self._title.split())

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

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

```

Invoking Class Methods

- We invoke methods on specific instances of our class
- In this example, we are invoking Book methods on specific Book objects

```
In [30]: # creating book objects:
         pp = Book('Pride and Prejudice', 'Jane Austen', 1813)
         emma = Book('Emma', 'Jane Austen', 1815)
         hp = Book("Harry Potter and the Sorcerer's Stone", "J.K. Rowling", 1997)

In [31]: hp.numWordsInTitle()

Out[31]: 6

In [32]: emma.yearsSincePub(2022)

Out[32]: 207

In [33]: hp.yearsSincePub(2022)

Out[33]: 25

In [34]: hp.sameAuthorAs(emma)

Out[34]: False

In [35]: emma.sameAuthorAs(pp)

Out[35]: True
```

Print Representation of an Object

```
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, its not helpful

- Special method **`__str__`** is automatically called when we ask to print a class object in Python
- **`__str__`** must always return a string
- We can customize how the object is printed by writing a custom **`__str__`** method for our class
- Very useful for debugging

`__str__` for Book class

- What is a useful string representation of a Book?
  - Something that combines the attributes in a meaningful way
  - The format( ) string method comes in handy here

```
# __str__ is used to generate a meaningful string representation for Book objects
# __str__ is automatically called when we ask to print() a Book object
def __str__(self):
    return "'{}', by {}, in {}".format(self._title, self._author, self._year)
```

  - Now when we ask to print a specific instance of a Book, we get something useful

```
In [21]: print(emma)
```

```
'Emma', by Jane Austen, in 1815
```

Summary

- Today we built a simple Book class
- **Declared attributes** using `__slots__`
- Briefly learned about about scope and naming conventions in Python
- Used the `__init__()` method to initialize Book objects with their attribute values
- Defined **accessor** and **mutator** methods to interact with attributes and avoid accessing attributes directly
  - Note about mutator methods: If an attribute cannot and should not change, no need to define a setter method for it!
- Implemented a few more "interesting" Book methods
- Implemented the `__str__()` method so that we get meaningful print statements for our Book objects