CS134 Lecture 23:
Classes and Objects III
Announcements & Logistics

- HW 7 due tonight (on Glow)
- Lab 8 is a partner lab: *autocomplete*
  - No prelab but do **read the handout** before arriving
  - Working with three classes, each in their own files
  - Good idea to use pencil/paper and map out the different attributes and methods
- Looking ahead: Lab 9 will be an implementation of the game **Boggle**
  - Brings together all OOP concepts, and we get to "build" a game

Do You Have Any Questions?
Last Time

- Built the Book **class** to represent book objects
- Learned about **private, protected, public** attributes and methods (indicate scope using underscores in Python)
- Explored **accessor** (getter) and **mutator** (setter) methods in Python
- Talked about **__init__** (aka constructor) and **__str__** methods
Today’s Plan

• Design a Library class that stores a sorted shelf of Book objects

• Tools we need:
  
  • `sorted()` function in Python (with optional parameter `key`)
  
  • requires us to pass a function as a parameter
  
  • first time using optional arguments in function/method calls

• We’ll also review some useful string methods, including:
  
  • `s.split()`, `s.join()`, `s.format()`
class Book:
    """This class represents a book with attributes title, author, and year"""
    # attributes: _title, _author, _year
        self._title = book_title
        self._author = book_author
        self._year = int(book_year)

    # accessor (getter) methods
    def get_title(self):
        return self._title
    def get_author(self):
        return self._author
    def get_year(self):
        return self._year

    # mutator (setter) methods
    def set_title(self, book_title):
        self._title = book_title
    def set_author(self, book_author):
        self._author = book_author
    def set_year(self, book_year):
        self._year = int(book_year)

    # methods for returning book properties
    def num_words_in_title(self):
        """Returns the number of words in title of book"""
        return len(self._title.split())
    def years_since_pub(self, current_year):
        """Returns the number of years since book was published"""
        return current_year - self._year
    def same_author_as(self, other_book):
        """Check if self and other_book have same author"""
        return self._author == other_book.get_author()
Library Class

- Let's build a Library class that stores a collection of Books
- Data attribute:
  - `_books` : collection of book objects
  - What built-in collection data type to use?
    - sorted, unsorted? mutable, immutable?
- What methods?
  - `__init__`, `__str__`
  - check out a book
  - return a book
- Invariant: shelves should remain in sorted order!
Library Class: Constructor

```python
from book import Book

class Library:
    '''Represents a sorted shelf of Book objects'''

def __init__(self, list_of_books=[]):
    self._books = [b for b in list_of_books]

if __name__ == "__main__":
    # creating book objects:
    b1 = Book('Pride and Prejudice', 'Jane Austen', 1813)
    b2 = Book('Emma', 'Jane Austen', 1815)
    b3 = Book("Parable of the Sower", "Octavia Butler", 1993)
    # creating library object
    lib = Library([b1, b2, b3])
```
from book import Book

class Library:
    '''Represents a sorted shelf of Book objects'''

def __str__(self):
    list_of_strings = []
    for book in self._books:
        list_of_strings.append(str(book))
    return " | ".join(list_of_strings)

if __name__ == "__main__":
    # creating book objects:
    b1 = Book('Pride and Prejudice', 'Jane Austen', 1813)
    b2 = Book('Emma', 'Jane Austen', 1815)
    b3 = Book("Parable of the Sower", "Octavia Butler", 1993)
    # creating library object
    lib = Library([b1, b2, b3])
    print(lib)
from book import Book

class Library:
    '''Represents a sorted shelf of Book objects'''

def checkout(self, title):
    '''given title (str) of a book, checks if it is in the library, if it is remove it and return True, else return False'''
    for book in self._books:
        if book.get_title() == title:
            self._books.remove(book)
            return True
    return False
from book import Book

class Library:
    '''Represents a sorted shelf of Book objects'''

def shelve(self, book):
    # add the book back to the shelves
    self._books.append(book)

    # now the shelves might be out of order!
    # lets sort them by author name
    self._books = sorted(self._books, key=Book.get_author)

To understand this, we need to review `sorted()` function in Python
Detour: Built-in `sorted()` function
**sorted()**

- **sorted()** is a built-in Python function (not a method!) that takes a sequence (string, list, tuple) and returns a *new sorted sequence as a list*

- By default, **sorted()** sorts the sequence in **ascending order** (for numbers) and alphabetical order for strings

- **sorted()** *does not alter the sequence* it is called on and always returns the type **list**

```python
>>> nums = {42, -20, 13, 10, 0, 11, 18} # set of ints
>>> sorted(nums) # this returns a list!
[-20, 0, 10, 11, 13, 18, 42]

>>> letters = ['a', 'c', 'z', 'b', 'Z', 'A']
>>> sorted(letters)
['A', 'Z', 'a', 'b', 'c', 'z']
```
Changing the Default Sorting Behavior

• To better understand the `sorted()` function, look at documentation

```python
help(sorted)
```

Help on built-in function sorted in module builtins:

```python
sorted(iterable, /, *, key=None, reverse=False)
```

Return a new list containing all items from the iterable in ascending order.

A custom key function can be supplied to customize the sort order, and the reverse flag can be set to request the result in descending order.

• An `iterable` is any object over which we can iterate (list, string, tuple, range)

• The optional parameter `key` specifies a function or method that determines how each element should be compared to other elements

• The optional boolean parameter `reverse` (which by default is set to `False`) allows us to sort in reverse order
Reverse Sorting Example

• Let’s consider the optional reverse parameter to sorted()

• Sort sequences in reverse order by setting this parameter to be True

  >>> nums = [42, -20, 13, 10, 0, 11, 18]

  >>> sorted(nums, reverse=True)

          [42, 18, 13, 11, 10, 0, -20]
Sorting with a **key** function

- Suppose we want to sort a data type based on our own criteria.
- Example: A list of **course tuples**, where the first item is the course name, second item is the enrollment capacity, and third item is the term (Fall/Spring).

```python
courses = [("CS134", 90, "Spring"), ("CS136", 60, "Spring"),
           ("AFR206", 30, "Spring"), ("ECON233", 30, "Fall"),
           ("MUS112", 10, "Fall"),   ("STAT200", 50, "Spring"),
           ("PSYC201", 50, "Fall"),   ("MATH110", 90, "Spring")]
```

- Suppose we want to sort these courses by their **capacity** (second element).
- We can accomplish this by supplying the `sorted()` function with a **key** function that tells it how to compare the tuples to each other.
- This same logic applies to sorting objects of any class that we define.
  - We can sort them based on a specific attribute.
Sorting with a **key** function

- **Defining a key function explicitly:**
  - We can define an explicit **key** function that, when given a tuple, returns the parameter we want to sort the tuples with respect to.

```python
def get_capacity(course):
    '''Takes a course tuple and returns capacity'''
    return course[1]
```

- We can pass this function as a **key** when calling `sorted()`

```python
# we can tell `sorted()` to sort by capacity instead
sorted(courses, key=get_capacity)
```
Sorting with a **key** function

- `sorted(seq, key=function)`
  - Interpret as `for el in seq: use function(el)` to determine where within sort order of `seq` that `el` belongs
  - For each **element in the sequence**, `sorted()` **calls the key function on the element** to figure out what “feature” of the data should be used for sorting

# we can tell sorted() to sort by capacity instead
```
sorted(courses, key=get_capacity)
```

- For each `course` in `courses` (a list of lists), sort based on value returned by `capacity(course)`
Example: Sorting with key

def get_capacity(course):
    '''Takes a course tuple and returns capacity'''
    return course[1]

# we can tell sorted() to sort by capacity instead
sorted(courses, key=get_capacity)

[('MUS112', 10, 'Fall'), ('AFR206', 30, 'Spring'), ('ECON233', 30, 'Fall'), ('STAT200', 50, 'Spring'), ('PSYC201', 50, 'Fall'), ('CS136', 60, 'Spring'), ('CS134', 90, 'Spring'), ('MATH110', 90, 'Spring')]
Sorting Objects using **key**

- Suppose we want to sort the Books in a list of Books using a specific data attribute (such as author's name)
- We can use the “getter” method for that attribute as our key argument
- Caveat: Key needs to be a **function** that can be applied to every object of the sequence, not a method that is called on an individual object
- Each method is a function that **belongs to a given class**
- The following are equivalent (left is method `get_author` called on Book `b`, right: function `Book.get_author` called on Book `b`):

```python
b = Book("Dune", "Herbert, Frank", 1965)

b1.get_author()  # b1 is a Book instance
Book.get_author(b1)  # Book is a class
```

```python
b1.get_author()  # Returns the author's name
Book.get_author(b1)  # Returns the same author's name
```
Sorting Objects using `key`

- The following sorts a list of Book objects by their author's name.
- To use the “getter method” from the class Book as key, we need to use the functional variant `Book.get_author`.
  - This function is called on every Book object and the result is used as the sorting criteria (author names).
- `sorted()` returns a new list of Book objects arranged in the alphabetical order of their author's name.

```python
sorted_books = sorted(list_of_books, key=Book.get_author)
```
Reading Books from CSV
Example in Class
Review: String Methods
Useful String Methods

Discover more str methods with `pydoc3 str`!

```python
>>> s = "    CSCI 134 is great!\n \t"
>>> s.strip()
'CSCI 134 is great!'

>>> lst = ['starry', 'starry', 'night']
>>> stars = '**'.join(lst)
>>> stars
'starry**starry**night'

>>> stars.split('**')
['starry', 'starry', 'night']

>>> "I have {} {} & {} {}".format(2, 'cats', 1, 'dog')
'I have 2 cats & 1 dog.'
```

- Remove whitespace from left/right sides of the string `s`
- Joins all elements from list of str `lst`, using the leading str `'**'`
- Splits all elements from str `stars`, using the str argument `'**'`
- Inserts arguments into the `{}` in the str instance object.
Special Methods
Special methods and attributes

- We’ve seen several “special” methods and attributes in Python:
  - `__name__` special module attribute
  - `__main__` name attribute of scripts
  - `__init__` method
  - `__str__` method
Other Special Methods

- There are many other “special” methods in Python.
  - `__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 in a few weeks!
Another Class Example
Another Example: Name Class

- Names of people have certain attributes
  - Almost everyone has a **first and last name**
  - Some people have one (or more) **middle name(s)**
- 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?
  - **initials**: can we define a method that returns the initials of people's names?
Example: Name Class

class Name:
    """"""Class to represent a person's name."""""

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

    def __str__(self):
        # if the person has a middle name
        if len(self._m) > 0:
            return self._f[0] + ' ' + self._m[0] + ' ' + self._l
        else:
            return self._f[0] + ' ' + self._l

>>> n1 = Name("John", "Schmidt", "Jacob Jingleheimer")
>>> n2 = Name("Paul", "Bunyan")

>>> print(n1)
J. J. Schmidt

>>> print(n2)
P. Bunyan
intials() method

• Suppose we want to write a method that returns the person’s initials as a string?
• How would we do that?
Example: Name Class

class Name:
    "'''Class to represent a person's name.''''

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

    def initials(self):
        if len(self._m) > 0:
            return self._f[0] + ' ' + self._m[0] + ' ' + self._l[0] + '.'
        else:
            return self._f[0] + ' ' + self._l[0] + '.

    def __str__(self):
        # if the person has a middle name
        if len(self._m) > 0:
            return self._f[0] + ' ' + self._m[0] + ' ' + self._l[0] + ' ' + self._l
        else:
            return self._f[0] + ' ' + self._l

>>> n1 = Name("John", "Schmidt", "Jacob Jingleheimer")

>>> n1.initials()
'J. J. S.'

>>> n2 = Name("Paul", "Bunyan")

>>> n2.initials()
'P. B.'