

CS 134 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
 - Good idea to use pencil/paper and map out the different attributes and methods
- Looking ahead: Lab 9 will be **Boggle**
 - Brings together all OOP concepts and get to "build" a game

Do You Have Any Questions?

Last Time

- Built the Book **class** to represents 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 and how to use **key** sorting
 - how to *pass a function* as an **argument** to another function
 - understand optional arguments in function/method calls
- Review some useful string methods:
 - **s.split(), s.join(), s.format()**

Last Time: Book Class

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

    # attributes: _title, _author, _year
    def __init__(self, book_title, book_author, book_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 (**`checkout`**)
 - return/add a book (**`shelve`**) and ensure shelf is sorted



Library Class: Constructor

```
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]
```

Create a **new list** containing the list of Book objects passed when an object is created

```
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])
```

Calls `__init__` on `lib` object (passed to `self`)

Library Class: `__str__`

```
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)
```

Calls `str` special method on each Book object and accumulates them in a list

```
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)
```

joins the string in `list_of_strings` together with the connector string " | " in between each

Calls `__str__` method on `lib` object

Library Class: Other Methods

```
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
```

List method that deletes the given item from the list

Library Class: Other Methods

```
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 author name
        self._books = sorted(self._books, key=Book.get_author)
```

To understand this, we need to review
sorted() function in Python

Default/Optional Arguments for Functions

Default/ Optional Arguments

- Sometimes we want to have optional input arguments for a function or have some arguments take default values
- Can do that by setting the default value in function definition

```
def function_with_optional_args(arg1, arg2, arg3=defval3):  
    '''optional arguments with default values always  
    come after the required arguments'''  
    # function body
```

Default Arguments: Example

- Sometimes we want to have optional input arguments for a function or have some arguments take default values
- Can do that by setting the default value in function definition

```
def greeting(name=""):
    '''Takes a name string (defaults to empty str)
    and prints a greeting.'''
    print("Hello " + name)
```

`greeting()`

Hello

`greeting("Maud")`

Maud

name is set to "Maud"

No name is passed, defaults to ""

Default arguments in Built-in Functions

- The optional/default arguments taken by built-in functions and methods are displayed when you query for its documentation
- Can do that by typing **help(type)** in Interactive Python or **pydoc3 type** in the Terminal

```
help(print)
```

```
✓ 0.0s
```

```
Help on built-in function print in module builtins:
```

```
print(...)
```

```
print(value, ..., sep=' ', end='\n', file=sys.stdout, flush=False)
```

```
Prints the values to a stream, or to sys.stdout by default.
```

```
Optional keyword arguments:
```

```
file: a file-like object (stream); defaults to the current sys.stdout.
```

```
sep: string inserted between values, default a space.
```

```
end: string appended after the last value, default a newline.
```

```
flush: whether to forcibly flush the stream.
```

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 (dictionary) order for strings
- `sorted()` **does not alter the sequence** it is called on and always returns the type `list`

```
>>> 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

```
help(sorted)
```

```
Help on built-in function sorted in module builtins:
```

```
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 criterion
- 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).

```
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

```
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()**

```
# 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 sort `seq`
 - 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

```
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')]
```

```
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
- Can use the getter method for that attribute and pass it to key
- Caveat: Key needs to be a **function** that is 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`):

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

`b1.get_author()` ↔ `Book.get_author(b1)`

Sorting Objects using **key**

- The following sorts a list of Book objects by their author's name
- Notice to use the getter method from the class Book as key
 - Need to use the functional variant **Book.get_author**
 - This function is called on every Book object which gives the sorting criteria (author names)
 - The return is a **list of Book objects** arranged in the alphabetical order of their author's name

```
sorted_books = sorted(list_of_books, key=Book.get_author)
```


Reading Books from CSV

Example in Class

Reading in CSV using String Methods

- Suppose we have a CSV `booklist.csv` with each line containing:
 - author name, title, year of publication
- We want to read this data and create a Library object containing corresponding Books
- Can use built-in string methods to process the lines



booklist.csv ×

Users > shikhasingh > Documents > gitrepos > courses > 134 > s24 > staff >

```
1 Orson Scott Card,Ender's Game,1985
2 Frank Herbert,Dune,1965
3 Douglas Adams,The Hitchhiker's Guide to the Galaxy,1979
4 Ray Bradbury,Fahrenheit 451,1953
5 George Orwell,1984,1949
6 Aldous Huxley,Brave New World,1932
```

Reading in CSV using String Methods

- Notice the use of accumulation variable that is a Library object and the built-in str methods

```
def process_books(filename):  
    '''Takes as input a CSV filename as string, returns  
    a Library object representing the books in the file.'''  
    new_lib = Library() # initialize to empty object  
    with open(filename) as book_info:  
        for line in book_info:  
            line = line.strip() # remove newline  
            author, title, year = line.split(',')  
            year = int(year) # convert year to int  
            new_lib.shelve(Book(title, author, year))  
    return new_lib
```

Review: String Methods

Useful String Methods

Find str methods: `pydoc3 str` (in Terminal) or `help(str)` in Notebook

```
>>> s = "    CSCI 134 is great!\n"
```

```
>>> s.strip()
```

```
'CSCI 134 is great!'
```

Remove whitespace from left/right sides of the `string s`

```
>>> lst = ['starry', 'starry', 'night']
```

```
>>> stars = '**'.join(lst)
```

```
>>> stars
```

```
'starry**starry**night'
```

Joins all elements from list of `str`, `lst`, using the leading `str '**'`

```
>>> stars.split '**')
```

```
['starry', 'starry', 'night']
```

Splits all elements from `str stars`, using the `str` argument `'**'`

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

```
'I have 2 cats & 1 dog.'
```

Inserts arguments into the `{}` in the `str` instance object

Summary

- Classes provide us with a way to further organize our code
- Methods are functions that belong to a given class and are called on instances of that class (using dot notation)
- Can store user-define types (Books) in Python built-in collections such as list, dictionaries, sets, etc
- Can sort any sequence containing built-in or custom types using **sorted**
- Optional/default arguments to functions: can define using **=default** in function definition, and can optional pass arguments during function call
 - Example: using key, reverse optional arguments in sorted
 - Default arguments in constructor (`__init__`)

Next Time: 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 `__`)