CS134 Lecture 23: Classes & Objects

Announcements & Logistics

- **Lab 6** graded feedback: almost done, will return soon
- **Lab 8** will be released today
  - Partner lab, no prelab
  - Focuses on creating and using our **own classes**
  - Will create our own ***autocomplete algorithm***
- **HW 7** due Mon 10 pm: focuses on understanding recursive code
  - Fewer questions but a little bit tricky

Do You Have Any Questions?

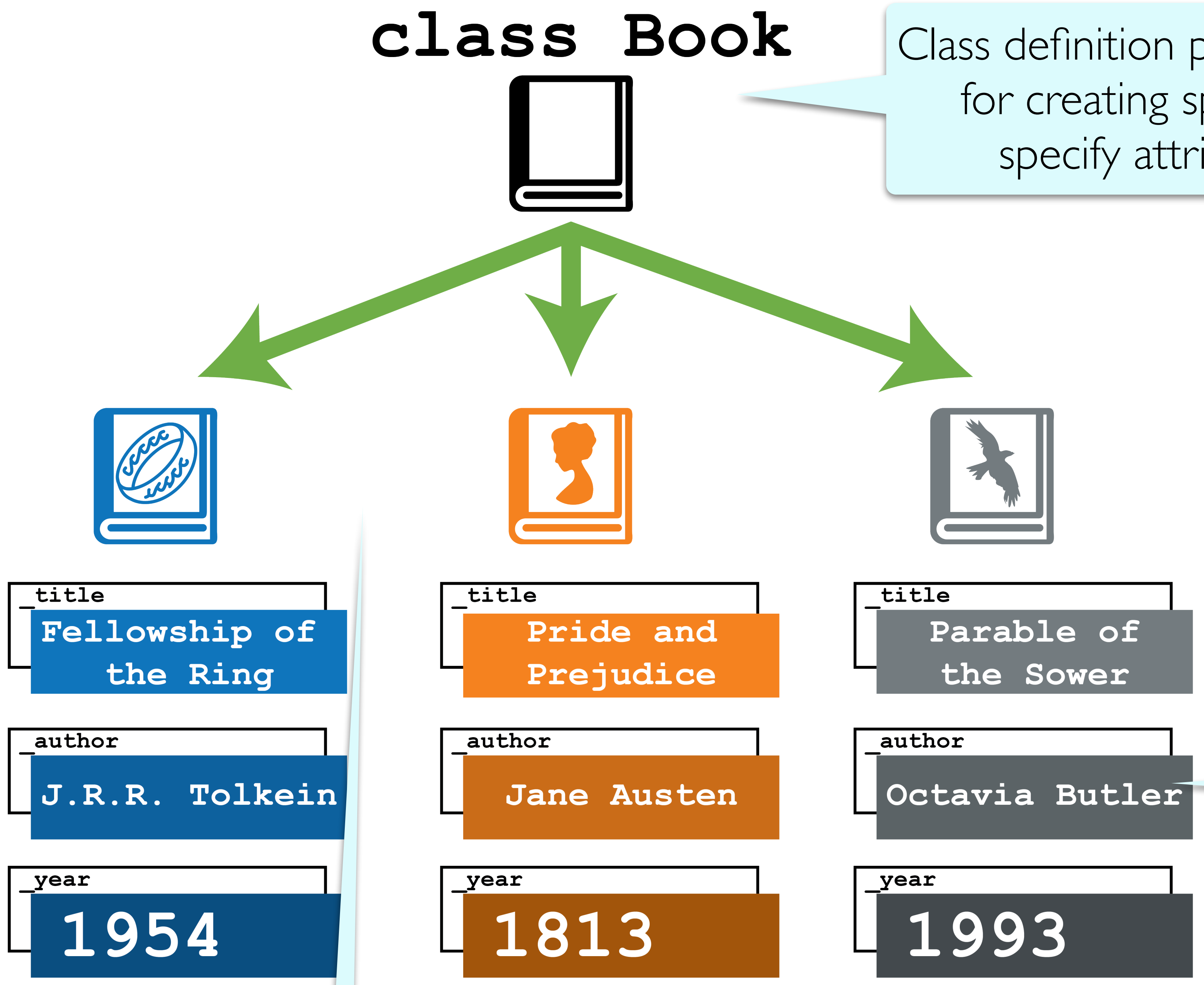
Last Time

- 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 how to define and call **methods** on objects of a **class**
    - first parameter in methods is always `self` (is a reference to the object that the method is called on)
- 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:
    - 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 Our Own Type: Book class



Class definition provides a "blueprint" for creating specific books and specify attributes of books

Providing values for attributes of the Book class, such as title, author, and year, define key features of individual instances

Specific instances of the Book class

Defining Our Own Class: Book

Name of class (always capitalized by convention)

```
class Book:
    """This class represents a book"""
    # indented body of class
    definition
```

Creating instances of the class:

```
book1 = Book()
book2 = Book()
```

book1 is an instance of class Book

book2 is another (different) instance of class Book

Attributes

- Objects have *state* which is typically held in **instance variables** or (in Pythonic terms) **attributes**
  - For the **Book** class, let's define attributes as
    - `_title, _author, _year`
    - the leading underscore in the variable name indicates that they are ***protected*** (these are not meant to used outside the class body)
- Every **Book** instance has different attribute *values*!
- In Python, we typically **declare** and **initialize** attributes in a special function known as the **constructor**
- The constructor has a special name: `__init__` and is typically defined at the top of the class before all other method definitions

Constructor: Defining `__init__`

```
class Book:
    """This class represents a book"""
    # attributes: author, title, year
    def __init__(self, book_author, book_title, book_year):
        self._author = book_author
        self._title = book_title
        self._year = book_year
```

Implicitly calls
`__init__(book1, "Alcott", "Little Women", 1869)`

Creating instances of the class:

```
book1 = Book("Alcott", "Little Women", 1869)
book2 = Book("Tolkein", "Lord of the Rings", 1954)
```

Class Methods

Methods and Data Abstraction

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

```
>>> # creating book objects
>>> ps = Book("Parable of the Sower", "Octavia Butler", 1993)
>>> ps._title
'Parable of the Sower'
```

- 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)

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


    # __init__ is automatically called when we create new Book objects
    # we set the initial values of our attributes in __init__
    def __init__(self, book_title, book_author, book_year):
        self._title = book_title
        self._author = book_author
        self._year = 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)
```

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

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


    # __init__ is automatically called when we create new Book objects
    # we set the initial values of our attributes in __init__
    def __init__(self, book_title, book_author, book_year):
        self._title = book_title
        self._author = book_author
        self._year = 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)
```

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

Using Accessor/Mutator Methods

```
>>> pp.get_title()
'Pride and Prejudice'
>>> emma.get_author()
'Jane Austen'
>>> ps.get_year()
1993
>>> ps.set_year(1991)
>>> ps.get_year()
1991
```

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)

Aside:
Naming Conventions in Python

Scope & Naming Conventions in Python

- Double leading underscore (__) in name (**strictly private**): e.g. `__value`
    - "Invisible" from outside of the class
    - Strong ***"you cannot touch this"*** policy (which is enforced)
- 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
    - **Most attributes in CS134 should start with a single underscore**
- No leading underscore (**public**): e.g. `value`
    - Can be freely used outside class
- These conventions apply to **methods names** and **attributes**

Attribute Naming Conventions

```python
class TestingAttributes():

    def __init__(self):
        self.__val = "I am strictly private."
        self._val = "I am private but accessible from outside."
        self.val = "I am public."
```

```
>>> a = TestingAttributes()
>>> a.__val
AttributeError: 'TestingAttributes' object has no attribute '__val'
>>> a._val
'I am private but accessible from outside.'
>>> a.val
'I am public.'
```

Note: Although we can access attributes directly using dot notation, it's bad practice: should always use methods to access/manipulate attributes

Class Methods: More!

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:
    - `num_words_in_title():` returns the number of words in the title of the book
    - `years_since_pub(current_year):` takes in the current year and returns the number of years since the book was published
    - `same_author_as(other_book):` takes another Book object as a parameter and checks if the two books have the same author

num_words_in_title()

- Returns the number of words in the title of the book

```python
class Book:
    ...

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

years_since_pub(current_year)

- Takes in the current year and returns the number of years since the book was published

```python
class Book:
    ...


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

same_author_as(other_book)

- Takes another Book object as a parameter and checks if the two books have the same author

```python
class Book:
    ...


    def same_author_as(self, other_book):
        """Check if self and other_book have same author"""
        return self._author == other_book.get_author()
```

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


    # __init__ is automatically called when we create new Book objects
    # we set the initial values of our attributes in __init__
    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()
```

Invoking Class Methods

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

```
>>> # creating book objects
>>> pp = Book("Pride and Prejudice", "Jane Austen", 1813)
>>> emma = Book("Emma", "Jane Austen", 1815)
>>> ps = Book("Parable of the Sower", "Octavia Butler", 1993)
>>> ps.num_words_in_title()
4
>>> emma.years_since_pub(2023)
208
>>> ps.years_since_pub(2023)
30
>>> ps.same_author_as(emma)
False
>>> emma.same_author_as(pp)
True
```

__str__ : special method called by print

Print Representation of an Object

```
class Book():

   def __init__(self, title):
      self._title = title

```

```
>>> test = Book("testing")

>>> print(test)

<__main__.Book object at 0x105eecca0>
```

By default, if we print an object, the output is 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

```python
# __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 "'"+self._title+"', by "+self._author+", in "+ str(self._year)
```

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

```
>>> print(emma)
'Emma', by Jane Austen, in 1815
```

Library Class: See Notebook

Aside: Built-in `sorted()` function

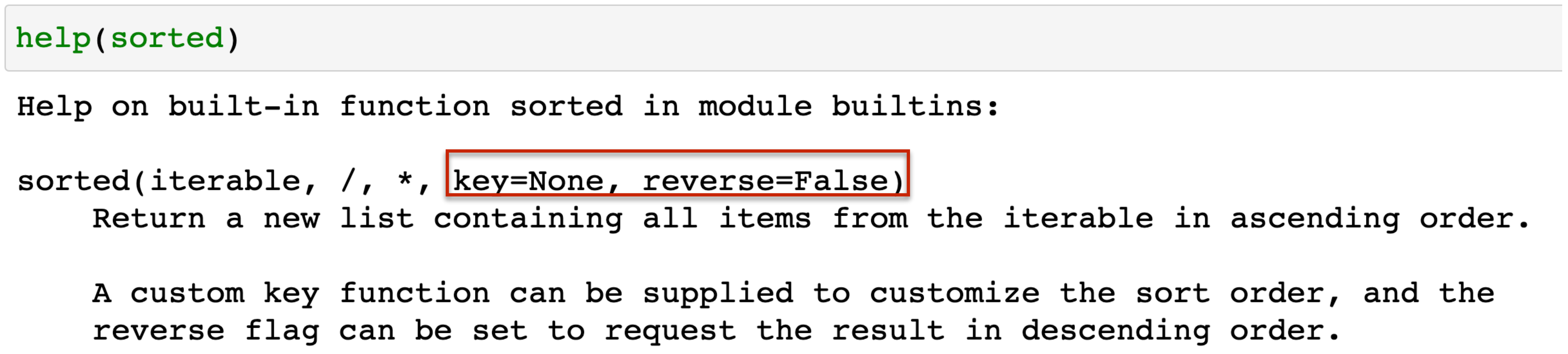
sorted( )

- `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 capacity(course_pair):
    '''Takes a sequence and returns item at index 1'''
    return course_pair[1]
```

  - Once we have defined this function, we can pass it as a `key` when calling `sorted()`

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

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

Sorting with a **key** function Example

```
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 capacity(course_pair):
    '''Takes a sequence and returns item at index 1'''
    return course_pair[1]
```

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

We will do a full lecture on **sorting** soon and discuss more about how it works in detail!

Other Special Methods

- There are many other "special" methods in Python.

`__eq__ (self, other):`	`x == y`
`__ne__ (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, and we can reimplement any of these for our class!

Summary

- Today we built a simple `Book` class
- (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 mutators: If an attribute 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