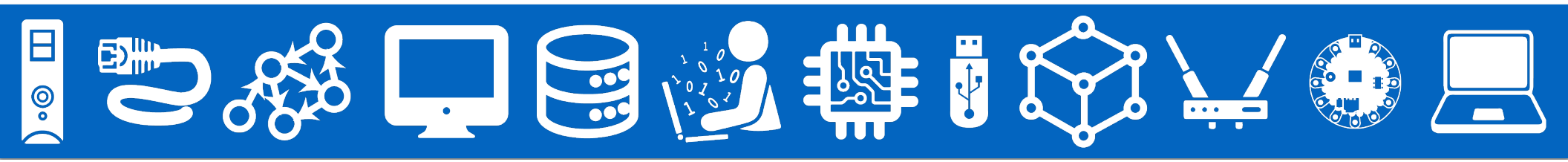
CS134:
Classes & Objects



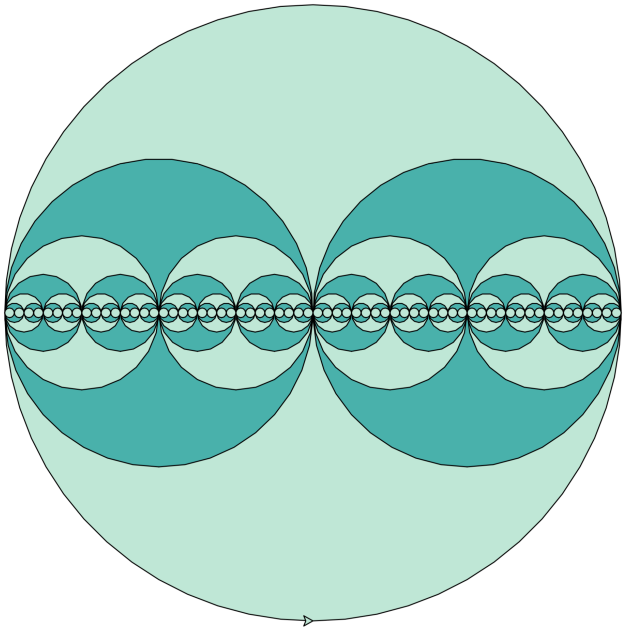
Announcements & Logistics

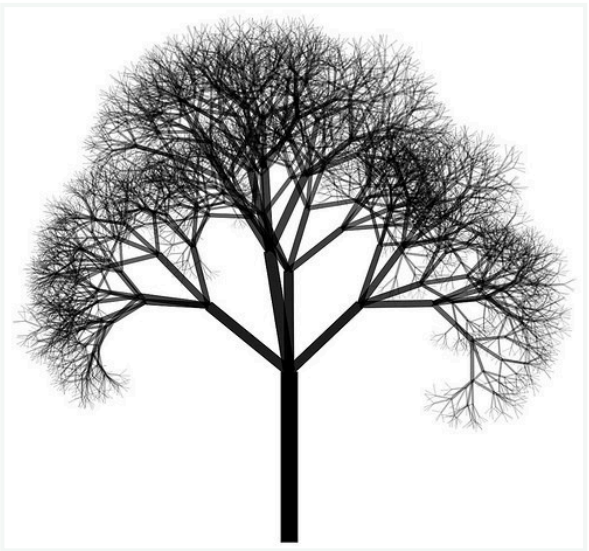
- **Lab 7** due Wed/Thurs, 10 pm
  - Make sure your images and values match the handout
- **HW 7** posted today, due Mon at 10 pm (on Glow)
- **Lab 8** is going to be a **partner lab**
  - Fill out partner google form (from Lida) by **tomorrow @ 10 pm**
  - **Both partners have to fill out the form!**
  - Can work by yourself but **strongly encourage** you to find a partner
  - **Must attend one lab session together**
  - Mon lab due on Wed, Tue lab due on Thur

Do You Have Any Questions?

Last Time

- Investigated a few more graphical recursion examples
- Wrapped up our recursion discussion
- Any remaining questions?

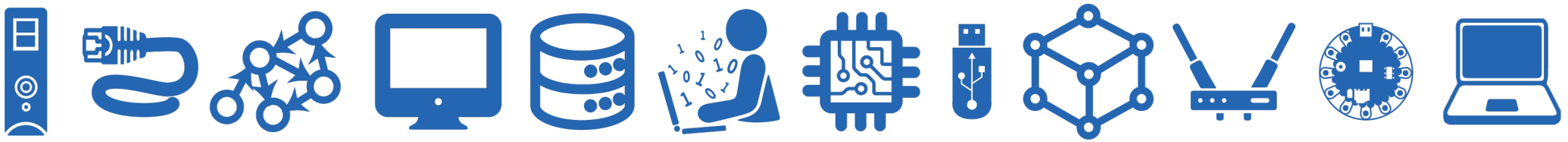




Today

- Start discussing our next topic: **classes** and **objects**
  - Python is an **object oriented programming** (OOP) language
  - Everything in Python is an **object** and has a **type**
- Learn how to define our own **classes** (**types**) and **methods**

Objects in Python



Objects in Python

- We have seen many ways to store data in Python

```
1234 3.14159  "Hello" [1, 5, 7, 11, 13] (1, 2, 3)
{"CA": "California", "MA": "Massachusetts"}
```

- Each of these is an **object**, and every object in Python has:
  - a **type** (int, float, string, list, tuples, dictionaries, sets, etc)
  - an internal **data representation** (primitive or composite)
  - a set of functions/methods for **interacting** with the object
- Vocab: A *specific object* is an **instance** of a type
  - `1234` is an instance of an **int**
  - `"Hello"` is an instance of a **string**

type(object)

- The `type()` function returns the data type for an object

```
>>> type(1234)
<class 'int'>
>>> type("hello")
<class 'str'>
>>> type([1, 5, 7, 11, 13])
<class 'list'>
>>> type(range(5))
<class 'range'>
```

Objects and Types in Python

EVERYTHING IN PYTHON IS AN OBJECT (AND HAS A TYPE)

- Even functions are a type!
- Guido designed the language according to the principle "first-class everything"

```
>>> def greeting():
...     print("Hello")
...
>>> type(greeting)
<class 'function'>
>>>
```

"One of my goals for Python was to make it so that all objects were "first class." By this, I meant that I wanted all objects that could be named in the language (e.g., integers, strings, functions, classes, modules, methods, and so on) to have equal status. That is, they can be assigned to variables, placed in lists, stored in dictionaries, passed as arguments, and so forth." — ***Guido Van Rossum***
(Blog, The History of Python, February 27, 2009)

Stepping Back: Object-Oriented Programming (OOP)

- Python is an **"object-oriented" language**
  - We have been hinting at this aspect all semester
  - Today we will embrace it!
- **OOP** (object oriented programming) is a fundamental programming paradigm
- It has four major principles:
  - **Abstraction** - handle complexity by ignoring/hiding messy details
  - **Inheritance** - derive a class from another class that shares a set of attributes and methods
  - **Encapsulation** - bundling data and methods that work together in a class
  - **Polymorphism** - using a single method or operator for different uses
- We'll explore some of these principles in more detail in the coming lectures

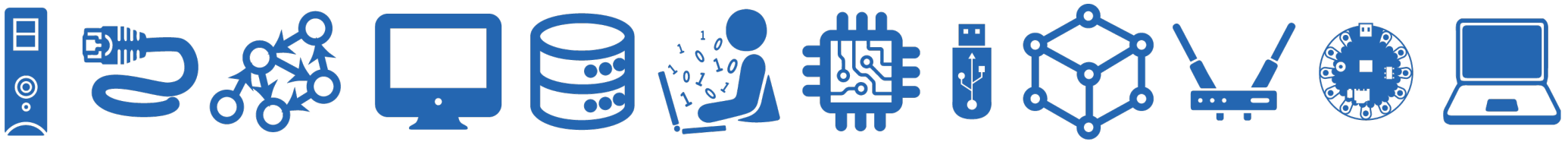
What are Objects?

- It's time to *formally* define **objects** in Python
- Objects are:
    - collections of data (variables or **attributes**) and
    - **methods** (functions) that act on those data
- Example of **abstraction**:
    - Abstraction is the art of hiding messy details
    - Methods define behavior but hide implementation and internal representation of data
        - Eg., You have been using methods for built-in Python data types (lists, strings, etc) all semester without really knowing how the methods are implemented

Example: `[1,2,3,4]` has type `list`

- We don't really know how Python stores lists internally
- Fortunately the typical Python programmer does not need how lists are stored to use list objects (we've been doing it all semester!)
- How do we manipulate lists? Using the methods provided by Python.
  - `myList.append()`, `myList.extend()`, etc.
- **Take away:** Internal representation of objects should be hidden from users. Objects are manipulated through associated **methods**.

Defining Our Own Types

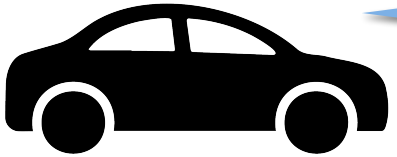


Creating Our Own Types: Classes

- It's time to move beyond just the built in Python objects!
- We can create our own data types by defining our own **classes**
  - Classes are like blueprints for objects in Python
- **Creating** a class involves:
  - Defining the **class name**, **attributes, methods**
- **Using** the class involves:
  - Creating **new instances** of the class (which create specific objects)
    - `myList = [1, 2],`
      `myOtherList = list("abc")`
  - Performing operations on the instances through methods
    - `mylist.append(3)`

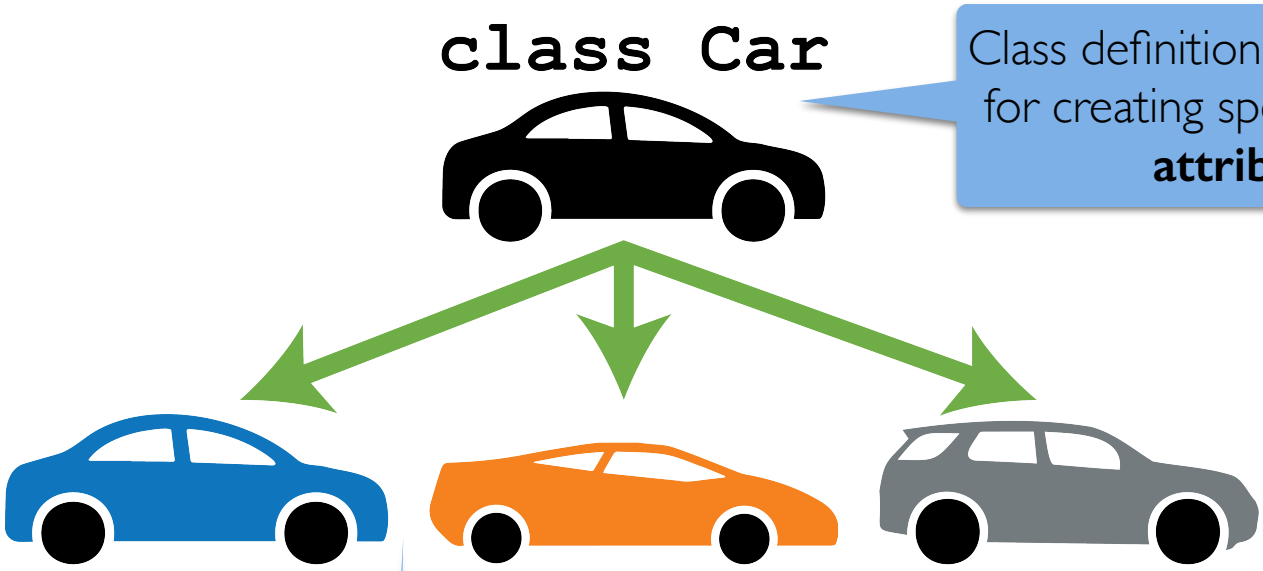
Defining Our Own Type: Car class

```
class Car
```



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

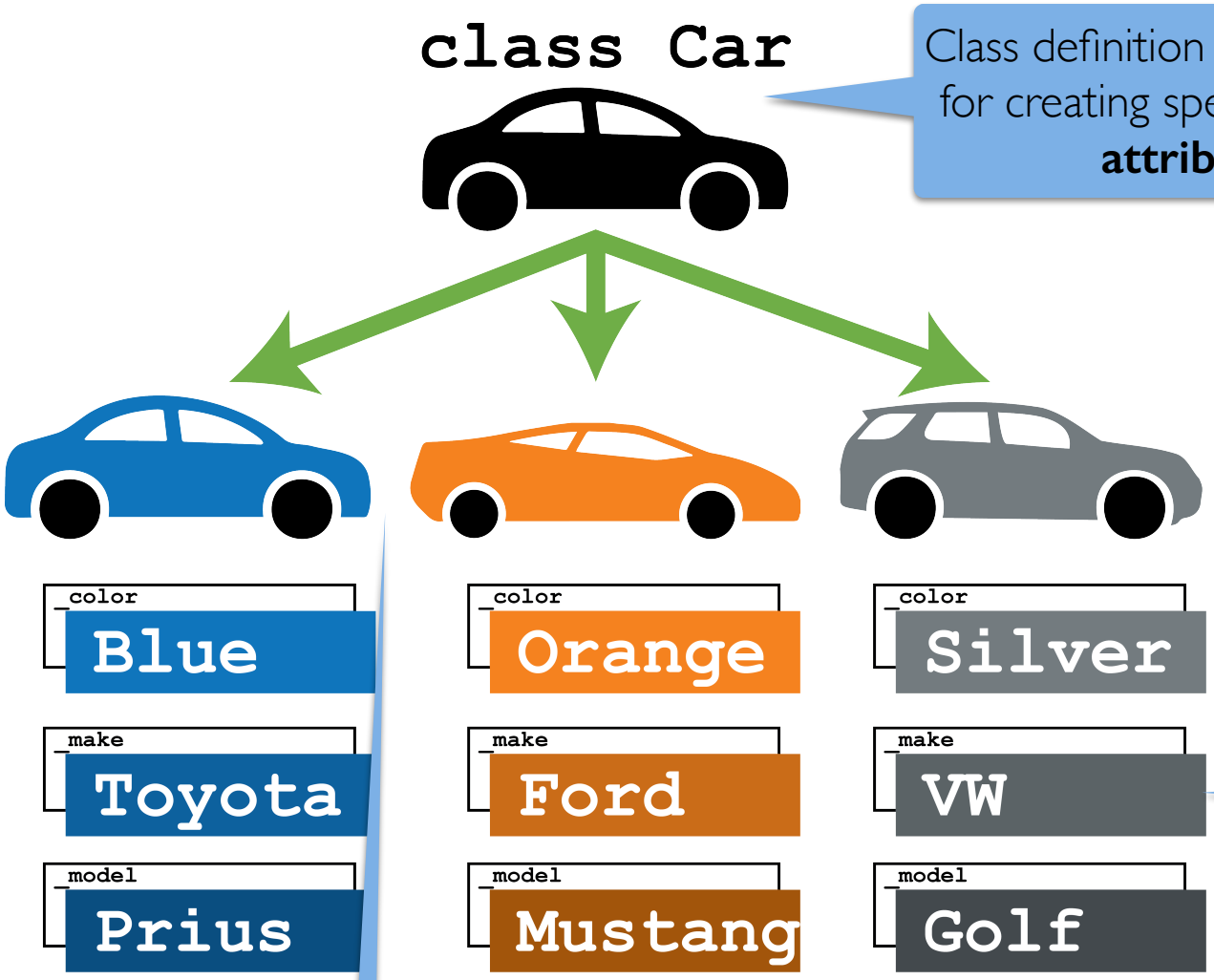
Defining Our Own Type: Car class



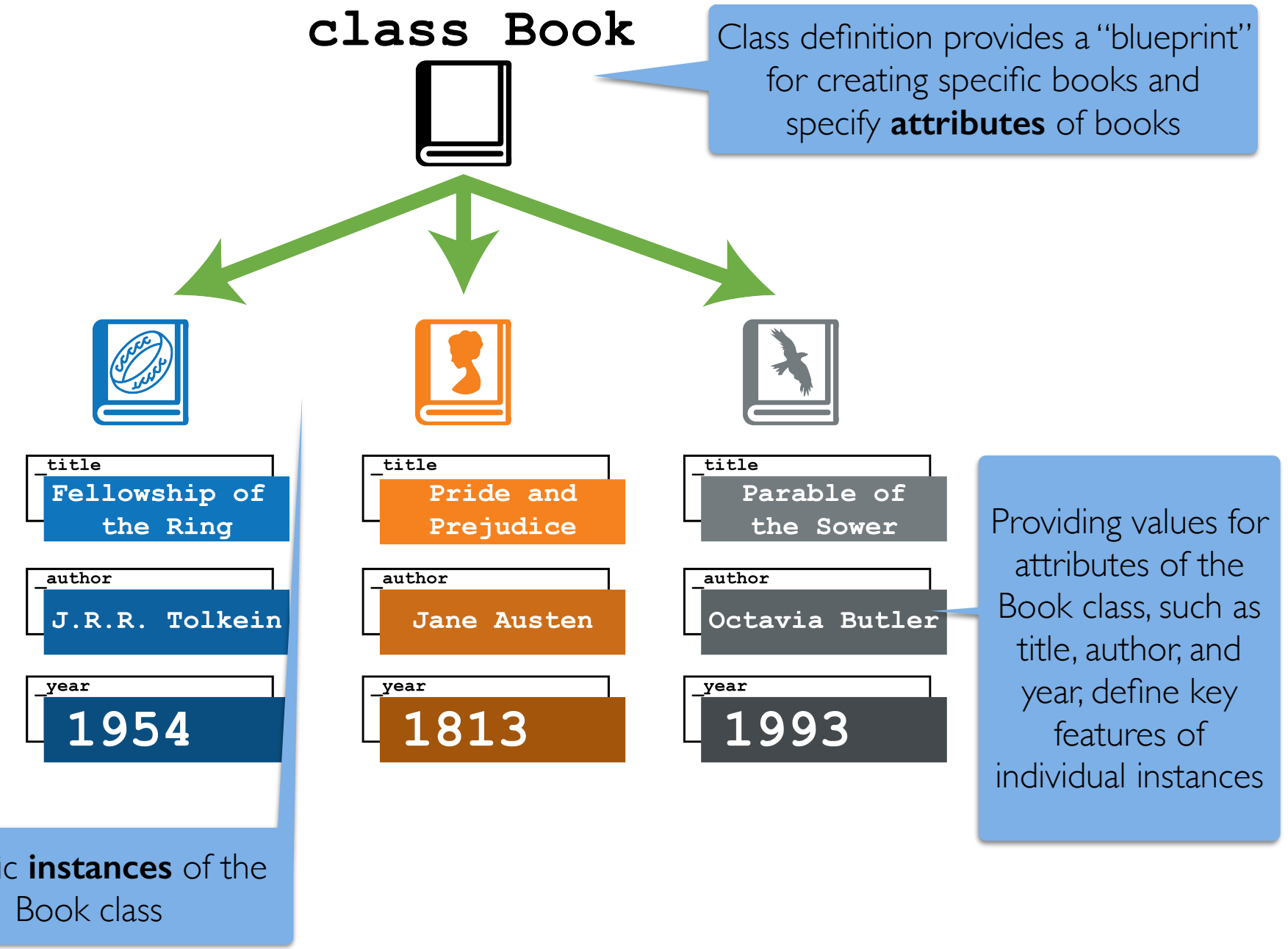
Class definition provides a "blueprint" for creating specific cars and specify **attributes** of cars

Specific **instances** of the Car class

Defining Our Own Type: Car class



Defining Our Own Type: Book class



Defining Methods of a Class

- Methods are defined as part of the class definition and describe how to interact with the class objects
- Example: Recall the following methods for the list class

```
>>> lst = list()
>>> lst.extend([1,2,3])
>>> lst
[1, 2, 3]
>>> lst.append(4)
>>> lst
[1, 2, 3, 4]
```

dot notation to "call" the method on the object

Defining Methods of a Class

- On the previous slide, we called methods like **`append()`** and **`extend()`** on a particular list **object** **`lst`**.
- We can define methods in our classes in a similar way
- Consider this simple example:

Class name (note the use of CamelCase by convention)

```
class SampleClass:
    """Class to test the use of methods"""
    def greeting(self):
        print("Hello")
```

Defining Methods of a Class

- To create methods that can be called on an instance of a class, they must have a parameter which takes the instance of the class as an argument
- In Python, the **first parameter of a method is always `self`, and is used as a reference to the calling instance**

```
class SampleClass:
    """Class to test the use of methods"""
    def greeting(self):
        print("Hello")
```

All methods include `self` as the first parameter.

Our First Method

```
class SampleClass:
    """Class to test the use of methods"""
    def greeting(self):
        print("Hello")
```

- How do we call the greeting method?
  - We create an **instance** of the class and call the method on that instance using dot notation:

```
>>> sample = SampleClass()

>>> sample.greeting()
Hello
```

sample is an instance of SampleClass

Invoke the `greeting()` method on sample

Mysterious `self` Parameter

- Even though method definitions have `self` as the first parameter, **we don't pass this parameter explicitly** when we invoke the methods
- This is because whenever we call a method on an object, the object itself is **implicitly** passed as the first parameter
- Note: In other languages (like Java) this parameter is implicit in method definitions but in Python it is explicit and by convention named **`self`**
- **Take away:**
  - When **defining** methods, always include **self**
  - When calling or invoking methods, the value for self is passed implicitly (meaning, we don't specify it, but it happens automatically)

Summary of Classes and Methods

- **Classes** allow us to define our own data types
- We create **instances** of classes and interact with those instances using methods
- All **methods** belong to a class, and are defined within a class
- A method's purpose is to provide a way to access/manipulate **instances** of the class)
- The first parameter in the method definition is **the reference to the calling instance (self)**
- When **invoking** methods, this reference is provided implicitly

Defining Our Own Class: Book

- 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 (**methods**)
- For example, suppose we want to define a new `Book` class
  - Attributes?
  - Methods?

Defining Our Own Class: Book

- 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 (**methods**)
- 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 (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**.
- 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

Next up

- So we have attributes and methods
- How do we assign values to attributes?
- What actually happens when we create a new instance of a class?
- More on this (and more!) next time

The end!