Classes
Attributes & Methods

6.100 LECTURE 19

SPRING 2026

Announcements

- Pset 5 due tonight
- Pset 6 released after class
- Pset 5 checkoff due next Tue 4/21
  - CPW this weekend, next Monday is holiday

- Friday office hours update
  - 3–5 pm in **38-370**
  - 5–7 pm in 36-153

Data Abstraction

Representing structured data

- General strategies
  - nested lists
  - parallel lists
  - nested dicts
  - parallel dicts
- Downsides
  - structure needs to be thoroughly documented
  - changing the representation can impact large amounts of code
  - forced to work with Python's built-in operations

Custom data types and attributes

- Python comes with built-in types
  - every object is an **instance** of a type
  - can write out a literal object, can also instantiate with `()`
- We can define our own types, using the `class` keyword
  - must instantiate with `()`
  - inspect object's type with `type()` or `isinstance()`
- An object of a custom type is basically a namespace
  - can store variables in it, pointing to other objects
  - these variables are called **attributes**
  - accessed via dot notation, ***`object.attribute`***
  - hence, objects are inherently mutable

Using functions in classes: bound methods

- Sometimes, it makes sense for functions to apply only to objects of a custom type
- Define those functions inside the class
  - a class is also an object, can store attributes inside it
  - now those functions can only be accessed through dot notation, ***`class.function(arg1, arg2, ...)`***
- Because these functions are meant to operate on instances of a class, would like to use dot notation on objects themselves
  - ***`obj.function(arg2, ...)`*** is equivalent and preferred to the ***`class.function()`*** syntax
  - ***`obj.function`*** is a **bound method,** not a real function, but automatically translated with ***`obj`*** as ***`arg1`*** when called

self convention

- While ***obj.function()*** is called without ***arg1***, ***function()*** still has to be defined with ***arg1*** parameter so it can be assigned to ***obj***
- Hence ***arg1*** is effectively always an object of type ***class***
- Widely accepted convention is to name ***arg1*** as **`self`**
- **`self`** has no inherent meaning in Python, just a name

`__init__()` method

- Tiresome to build objects by manually specifying attributes
- Python recognizes certain "double-underscore" (or "dunder") method names
- When a class is called to instantiate a new object
  - an empty object is created
  - that object is passed to **`__init__()`** if it exists
  - any arguments passed to the ***`class`*`()`** call also get passed to **`__init__()`**
  - the body of **`__init__()`** can choose to populate the object (i.e., **`self`**) as desired
  - the now populated/initialized object is returned
- **`__init__()`** fills a similar role as "constructor" in other languages
  - mechanism is somewhat different

`__str__()` method

- Another “dunder” method
- When call **`str(`*`obj`*`)`**, automatically defers to ***`obj`*`.__str__()`** if it exists
- When call **`print(`*`obj`*`)`** or evaluate **`f"{`*`obj`*`}"`**, also automatically retrieves result of **`str(`*`obj`*`)`**
- Customize the string representation of your classes and how they print
  - useful for debugging
  - recommend implementing **`__str__()`** right after **`__init__()`**

Non-method class attributes

- Recall that functions in classes are just attributes
- Can define other attributes that don't point to function objects
- These are typically used to store information common to the class and not specific to any one object instance
- A common use case: Keep track of ID number generation, so that `__init__()` can assign each new object a unique ID

Protecting internal state

- Python has no watertight mechanism to prevent modification of an object's attributes
- **Mitigation 1:** when storing mutable data passed into **`__init__()`**, consider **storing a copy**
- **Mitigation 2:** specify **accessor methods** (getters and/or setters)
  - encourage users to follow an **interface**
  - use ***`obj._leading_underscore`*** convention to discourage direct attribute access
  - same principle applies to class attributes

Next time

- **Pset 6** is an extension of last Wednesday's lecture, and also asks you to implement and use a class
- No class next Monday
- Next Wedesday
  - extend classes with **inheritance**
  - **Pset 7** will exercise
- This Wednesday
  - detour on **exceptions** and **assertions**