Classes

Special Methods & Inheritance

6.100 LECTURE 21

SPRING 2026

Announcements

- Pset 6 due this Friday 4/24
  - checkoff due next Friday 5/1
- Pset 7 released today after class

- More examples of using classes next week
- Exam 3 in 1.5 weeks, Monday 5/4
- Pset 7 due after exam on Wednesday 5/6
  - checkoff due on last day of classes Tuesday 5/12

- Lectures and recitation after Exam 3 preview topics in CS curriculum

Last time

- **`Student`** class
  - a class is an object of type **`type`**
  - defines functions that operate on **`Student`** objects
    - **`__init__()`**
    - **`calculate_total()`**
    - **`print_record()`**
  - convention of **`self`** as first formal parameter name
  - function names are stored as class attributes
- **`Student`** instances
  - contains attributes initialized by **`__init__()`**
  - access **`Student`** functions/operations as method calls
    - **`tony.`*`operation`*`(...)`** gets translated to **`Student.`*`operation`*`(tony, ...)`**
    - in function call frame, **`self`** gets bound to the **`tony`** object

Class and instance attributes

- **Class** and **instance objects** are just "namespaces"
  - they live on the heap, **mapping attributes (names) to other objects** on the heap
  - very similar structure as modules
    - e.g., import math, random, matplotlib, etc.
  - do NOT confuse them with **function call frames,** which live on the stack **only when a function call is active**
- **Class attributes**
  - usually point to functions/methods
  - but can also maintain class-specific data, e.g., ID counters
- **Instance attributes**
  - usually set in **`__init__()`** and accessed throughout methods
  - if name begins with a **single underscore,** indicates not intended for use outside the class
    - particularly useful for "hiding/protecting" mutable attributes from outside access

Other double-underscore methods

- We have already seen **`__init__()`** and **`__str__()`**
  - **`__init__()`** is implicitly called when creating new instances
  - **`__str__()`** is implicitly called when converting with **`str()`**
- Fraction example
  - Enable **`float(fraction)`** float with **`__float__()`**
  - Enable **`*`** operator with **`__mul__()`**
  - Enable **`/`** operator with **`__truediv__()`**
    - can we reuse **`__mul__()`**?
  - Enable **`==`** operator with **`__eq__()`**
    - how to recognize **`Fraction(-1, 4) == Fraction(3, -12)`**?
- Listings of special names
  - https://docs.python.org/3/reference/datamodel.html#special-method-names
  - https://docs.python.org/3/reference/datamodel.html#basic-customization
  - https://docs.python.org/3/reference/datamodel.html#emulating-numeric-types

Inheritance

Inheriting from `object`

- Every Python type inherits from **`object`**
  - already provides **`__init__()`**, **`__str__()`**, **`__eq__()`**
  - so we can create, print, compare
  - but default behavior may not be what we want
    - **`object.__str__()`** prints memory address
    - **`object.__eq__()`** compares type and memory address
  - so when we define them in our own classes, **they override access to `object`'s attributes**

Inheritance rules for attribute access

```
print(critter.age)
```

- When looking up attribute on an instance object
  - **If the attribute name is in the instance,** evaluate to the object it references
    - remember: this can evaluate to any **object** in memory, but NOT a name/attribute
  - If not in the instance, **look in that instance's class**
    - this is how method lookups work
    - also works on any class attribute
  - **If not in the class object, look in that class's parent class**
  - Etc...
- The above is applicable only for **evaluating an attribute**
  - When **setting an attribute** for an instance, the attribute is set directly inside the instance, even if the attribute name exists in the class hierarchy

```
critter.size = "tiny"
```

Using inheritance

- Subclasses can **override methods** of their parent/superclass
  - We already know how to override **object**'s **`__init__()`** and **`__str__()`**
  - **`Cat`** overrides **Animal**'s **`__str__()`**
  - **`Cat`** also overrides **`Animal`**'s **`speak()`**, providing a working implementation
- Subclasses can **reuse methods** of their parent/superclass
  - **`Cat`** relies on **`Animal`**'s **`__init__()`**
  - **`Cat.confuse()`** relies on **Animal**'s **`get_age_diff()`**
- Same applies to class and instance **attributes**
  - subclass instances can rely on attributes initialized in superclass **`__init__()`**
  - but **risky** if superclass implementation changes, consider using **getters/setters**

Retaining and extending superclass functionality

- Sometimes, want to preserve superclass method behavior while extending it
  - simply overriding it would require code duplication
- Strategy: superclass methods still available through explicit ***`superclass.method`*`(self, ...)`** reference
  - because not accessing as object's method, requires passing in the object (usually **`self`**) as first argument
- ***Inside a subclass method's body,*** can also use **`super()`** to reinterpret **`self`** as an instance of the superclass
  - **`Animal.__init__(self, age, name)`**
  - **`super().__init__(age, name)`**
  - differing opinions on which is better, but **`super()`** is common

Next time

- No significant new content/concepts going forward
- More examples of using classes next week and on Pset 7