# Classes

## Special Methods & Inheritance

6.1000 LECTURE 15

FALL 2025

# Announcements

- Pset 4 due Wed 10/29
- Pset 5 released Wed 10/29, due next Fri 11/7
- Midterm 2 the following Wed 11/12

- Note about https://pythontutor.com/
  - doesn't display classes and instance objects exactly the same as we do
  - but same concepts are shown
- **Read the slides! Experiment with the code!**

# 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`** objects/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 attributes

- Attributes stored in classes can point to any type of object, not just functions
- Common use case: manage unique IDs for instances
  - maintain a class "counter" or "set"
  - when initializing a new instance (in **`__init__()`**), set an instance **`id`** attribute based on state of class attribute
  - update class attribute for next instance initialization

# Single underscore convention

- Python has no real mechanism to protect data/attributes from access
  - given reference to an object, can see that object's attributes
- Common pitfall: aliasing of mutable attributes through **`__init__()`**
  - good practice to consider making copies
  - all the usual considerations of aliasing apply
- Common convention: indicate attributes not intended for public access with **single leading underscore**
  - define an **interface of operations** instead
  - access their info via **getter/setter methods**
- **Be able to work with objects without relying on direct attribute access**

# Double-underscore methods

- **`__str__()`** is a "double underscore" or "dunder" method
- When call **`str(`*`obj`*`)`**, automatically maps 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__()`**

# Other double-underscore methods

- We have already seen **`__init__()`**
  - implicitly called when constructing a new instance via **`Student()`**
- 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

## (at last!)

# 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 functionality while extending it
  - simply overriding it would require code duplication
- Strategy: superclass methods still available through explicit ***`superclass.method()`*** 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

# Designing `__eq__()`

- **`Rabbit`** example
  - each **`Rabbit`** has two parents and a unique ID
  - new **`Rabbit`**s are created by **`+`**
  - want siblings from the same parents to compare **`==`**
- Version 1
  - compare **`==`** on parent tuples
  - triggers **`==`** and hence **`__eq__()`** on elements, recursion!
  - if self is a **`Rabbit`** and other is **`None`**, then invalid to access **`other.parents`**
- Version 2
  - directly compare parent IDs
  - avoids recursion, saves computation
  - runs into same problem retrieving **`parent.id`** if **`parent`** is **`None`**
- Version 3
  - so close! need a valid "ID" for a **`None`** parent
  - wrap that concept in a helper function
  - good opportunity to use **`lambda`**