* Course Overview
  + Python is an object oriented programming language
* Overview
  + Naming special functions
    - \_\_feature\_\_
    - Called dunder [feature]
  + Dunder
    - Portmanteau of ‘double underscore’
  + Instance attributes
    - Defined in \_\_init\_\_
    - Assigned on per object basis
  + Ex)
    - class Rectangle:
    - def \_\_init\_\_(self, width, height)
    - self.width = width
    - self.height = height
* Class Attributes
  + Attribute that is associated with a class but not with each instance of a class
    - An attribute whose valued is shared between all instance of a class
  + Ex)
    - class MyClass:
    - my\_class\_attriute = “class attributes go here”
    - MY\_CONSTANT = “they are often class-specific contants”
    - def \_\_init\_\_(self):
    - self.my\_instance\_attribute = “instance attributes here”
  + Scopes in Python
    - Local: Inside the current function
    - Enclosing: Inside enclosing functions
    - Global: At the top level of the module
    - Built-in: In the special builtins module
  + To access class attributes you have to use ClassName.class\_attribute
  + Ex)
    - def \_\_init\_\_(self, …):
    - ….
    - ShippingContainer.next\_serial += 1
  + The Zen of Python: Explicit is better than implicit
  + Can access the class attribute outside the class with ClassName.class\_attribute
    - Can also access the class attribute through an instance
  + Ex)
    - ShippingContainer.next\_serial
    - c4.next\_serial
  + can also access class attributes inside a class using self.class\_attribute
  + ex)
    - def \_\_init\_\_(self, …):
    - ….
    - self.next\_serial += 1
  + Best to avoid self and use ClassName.class\_attribute
    - Make is clear if instance attribute and which are class attribute
  + Pitfall: although you can read class attribute through self reference, attempting to assign to a class attribute through self reference won’t have the desired effect
    - Assigning to an instance attribute is how you bring the attribute into being
    - self.class\_attribute will create an instance attribute that shadows the actual class attribute
  + There is no class scope in python
* Static Methods
  + \_method(): are implementation detail
    - Not intended for use out side
  + \_\_method\_\_(): are special methods
  + There are to two ways to associate method with the class rather than instances of the class
    - Static method decorator
  + Ex)
    - @staticmethod
    - def \_generate\_serial():
    - ….
  + Static methods are decorated with @staticmethod and they don’t require the self parameter
    - Use ClassName.\_static\_method\_name() to call static methods
    - Self can still be used to call static methods but explicit is better than implicit
  + The Static Terminology is a relic from c and c++
* Class Methods
  + Use decorate @classmethod, method accepts cls as first argument
  + Ex)
    - class MyClass:
    - attribute = “class attribute”
    - @classmethod
    - def my\_class\_method(cls, message):
    - cls.attribute = message #access class attribute via cls
  + cls plays an analogous role to self
  + @classmethod
    - Requires access to the class object to call other methods or the constructor
  + @staticmethod
    - No access needed to either class or instance objects
    - Most likely an implementation detail of the class
    - May be able to be moved outside the class to become a global-scope function in the module
  + The ‘named constructor’ idiom
    - A factory method which returns an instance of a class
    - The method name allows callers to express intent, and allows construction to be performed with different combinations of arguments
    - Originally a C++ idiom, also applicable in python
  + Ex)
    - @classmethod
    - def create\_empty(cls, owner\_code):
    - return cls(owner\_code, contents=[])
  + This allows us to support multiple constructors with different behaviors
* Static Methods with Inheritance
  + to get polymorphic override, we need to call static method on an instance
  + ex)
    - ……
    - self.bic = self.\_make\_bic\_code(....)
  + when you call static method through the class you prevent override being invoked
  + For polymorphic dispatch invoke static methods through self
* Class Methods with Inheritance
  + calling parent method in subclass
  + ex)
    - def \_\_init\_\_(self, owner\_code, contents, celsius):
    - super().\_\_init\_\_(owner\_code, contents)
    - …
  + use \*\*kwargs to capture parameters that may or may not be there
  + void circular dependencies
    - base classes should have no knowledge of subclasses
  + use \*\*kwargs to thread arguments through named-constructor class-methods to more specialized subclasses
* Properties
  + can maintain class invariant using helper methods, considered no pythonic
  + ex)
    - def get\_celsius(self): ….
    - def set\_celsius(self): …
  + better to encapsulate getter and setter methods in properties which behave like attributes
    - by using @property
  + ex)
    - @propety
    - def celsius(self):
    - return self.\_celsius
  + to call that property do example.celsius
  + @property allows you to call getter methods so they can be called as if they are attributes
    - attempting to assigning to it will return an AttributeError
  + decorator accepts functions as an argument and return a object, usually a wrapper, around the original function that modifies its behavior in some way
  + for setter functions use @p.setter, p is a stand in for the same name as the getter
  + ex)
    - @p.setter
    - def p(self, value):
    - self.\_p = value
  + ex)
    - @property
    - def celsius(self):
    - return self.\_celsius
    - @celsius.setter
    - def celsius(self, value):
    - if value > RefrigeratedShippingContainer.MAX\_CELSIUS:
    - raise ValueError(“Temperature too hot!”)
    - self.\_celsius = value
  + allows for self encapsulation
    - uses of internal attributes uses getter and setter rather than directly accessing the underlying attribute
  + too many properties can lead to excessive coupling
  + Tell! Don’t ask.
    - tell other objects what to do instead of asking them their state and responding to it
* Properties and Inheritance
  + to override a property getter, redefine in a derived class
    - delegate to base class via super() if we need to
  + to override setter, have to fully qualify getter from base class
  + ex)
    - @RefrigeratedShippingContainer.celsius.setter
    - def celsius(self, value): …
  + property object produce by property decorated keeps references to getter and setter, get and fset
* Overriding Properties with Tem…
  + Template method design pattern
  + ex)
    - class AbstractClass:
    - def template\_method(self): #operation defined in terms of as yet undefined steps
    - self.\_part1()
    - self.\_part2()
    - def \_part2(self):
    - raise NotImplementedError(“Override this method”) #abstract sub-operations must be overridden
    - #or may have default implementations, which may be overridden
    - class ConcreateClass(AbstractClass):
    - def \_part1(self):
    - print(“About to perform actions”)
    - def \_part2(self):
    - perform\_action() #concrete class fills in the details by overriding methods from the abstract class
  + don’t override properties directly
  + delegate to regular methods and override those instead
* Summary
  + class attribute versus instance attributes
  + class attributes are shared between instances
  + navigate to class attributes via the class
  + assigning to self always creates an instance attribute
  + use @staticmethod for methods which need neither the class nor the instance
  + use @classmethod for methods which needs the class but not the instance
  + use @classmethod for the named-constructor idiom
  + static and class methods can be overridden
  + static and class methods are polymorphic when invoked through self
  + use the @property decorate instead of getters and setters
  + easily override properties by delegating to regular methods
* Overview
  + three built-in functions for string representation of objects
    - r = repr(obj)
    - s = str(obj)
    - f = format(obj)
  + customization gives
    - maintainability
    - debuggability
    - usability