* 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
  + latitude: displacement north from the equator at 0
    - 1 to 90 is in the northern hemisphere
    - -1 to -90 is in the southern hemisphere
  + Longitude: displacement east from the prime meridian
    - 1 to 180 are in the east
    - -1 to -180 are in the west
  + repl(object)
    - returns string [containing module].[type] object at [hexadecimal address in memory]
  + str(object)
    - same as repr
  + format(oslo)
    - same as repr
  + these three are inherited from object base class that all class implicitly inherits
  + dir(object): list all attribute and methods of object
* Customizing repr()
  + Define \_\_repr\_\_(self) in class to override
    - Only thing is has to do is return a string
  + Ex)
    - def \_\_repr\_\_(self):
    - return f”Position {self.latitude} {self.longitude}”
  + Consider the target audience
    - repr() is intended for us
  + when python repl needs to display the result of an expression it request the repr of the result
    - ex)
    - repr(syndey) => ‘Poistion …’
    - sydney > Position ..
  + Good convention for Good \_\_repr\_\_ results
    - Include necessary state but be prepared to compromise
    - Format as constructor invocation source code
  + Ex)
    - def \_\_repr\_\_(self):
    - return f”Position(latitude={self.latitude}, longitude{self.longitude})”
  + eval(string): evaluate string as if it was source code
  + don’t want to hard code name of class into \_\_repr\_\_ return string
    - replace hard coded class name with expression that returns the same name
  + ex)
    - return f”{self.\_\_class\_\_.\_\_name\_\_}(latitude={self.latitude}, longitude={self.longitude})”
  + class of object is same as the type of an object, so you can use built in type() instead of retrieving class attribute directly
    - better to use built in functions over direct access when possible
  + ex)
    - return f”{type(self).\_\_name\_\_}(latitude={self.latitude}, longitude={self.longitude})”
  + can also create helper function to return type.\_\_name\_\_
  + ex)
    - def typename(obj):
    - return type(obj).\_\_name\_\_
    - def \_\_repr\_\_(self):
    - return f”{typename(self)}(latitude……)
  + For empty blocks use pass
    - Python doesn’t allow empty blocks, so use pass
  + Ex)
    - class EarthPosition(Position):
    - pass
  + The default \_\_repr\_\_ inherited from object is not much u se
  + Override \_\_repr\_\_ to return a more useful string, which ideally formatted as source code for a constructor call
  + You should almost always override \_\_repr\_\_() for you classes
* Customizing str()
  + str() function
    - the string class is callable
    - str(obj)
  + str(obj) invokes obj.\_\_str\_\_()
  + \_\_str\_\_() invokes \_\_repr\_\_ by default
  + Override \_\_str\_\_ to specialized behavior
  + str() is intended for system consumers
  + print() also uses \_\_str\_\_ to display objects
* Customizing format()
  + Override \_\_format\_\_() to specialize behavior
  + \_\_format\_\_ invoke \_\_str\_\_() by default
  + \_\_format\_\_ accepts self and format\_spec as parameters
  + Ex)
    - def \_\_format\_\_(self, format\_spec):
    - return “FORMATTED POSITION”
  + F strings and string.format() delegate to \_\_format\_\_
  + format\_spec
    - format specifier
    - string which controls how first argument will be formatted
    - what is allowed is dependent on the type of argument
  + floats
    - format(q, “f”): fixed point representation without e format
    - format(q, “.[num]f”) display with num precision
  + can use format specify in f strings
  + ex)
    - f”The conductance quantum is {q:.6f}”
  + convention that default format should return same about as \_\_str\_\_
    - have \_\_str\_\_ call format
  + ex
    - def \_\_str\_\_(self):
    - return format(self)
* Summary
  + sepr() gives a string for developers
  + str() the string constructor, gives a string for users
  + format() gives more control then str()
  + repr(obj) delegates to obj.\_\_repr\_\_()
  + str(obj) delegates to obj.\_\_str\_\_()
  + format(obj, spec) delegates to obj.\_\_format\_\_(spec)
  + all classes inherit default \_\_repr\_\_(), \_\_str\_\_(), and \_\_format\_\_()
  + most classes should override \_\_repr\_\_()
* Review: Single Inheritance
  + Single inheritance
    - class SubClass(BaseClass):
    - subclass inherit all attributes
    - may override methods
  + correct initialization
    - it generally necessary to call base class initializers to ensure proper object initialization
  + without override
    - if a subclass doesn’t define an initializer, then the base class initializer is called during construction
  + unlike other languages, python doesn’t automatically call base class initializers
  + \_\_init\_\_ is treated just like any other method
  + If a subclass defines \_\_init\_\_, it must explicitly call the base class implementation for it to be run
* Type Inspection
  + isinstance()
    - determines if an object is an instance of type
    - takes an object as its first argument and a type as its second
    - returns True if the first argument is an instance of the second
    - returns True if first argument is a sub class of the second
  + checking multiple types
    - isinstance can be used to check multiple types
    - isinstance(obj, (type\_a, type\_b, type\_c))
    - is obj is instance of any type of the tuple
  + isinstance() can be used for type checking in Python
  + some people consider type checking a sign of poor design
  + sometimes they’re the easiest way to solve a problem
  + issubclass()
    - operates on types to check for sub/superclass relationships
    - operates on types only instead of instances
    - determines if one class is a subclass of another
    - takes two arguments, both of which must be types
    - returns True if the first argument is a subclass of the second
* Multiple Inheritance
  + Defining a class with more than one direct base class
  + Not universal among object-oriented languages
  + Can lead to certain complexities
    - Such as when base class contains similar methods
    - Python has a relatively simple system for dealing with them
  + Multiple inheritance syntax
    - class SubClass(Base1, Base2, Base3):
  + Name resolution with multiple base classes
    - Classes inherit all methods from all of their bases
    - It there’s not method name overlap, names resolve to the obvious method
    - In the case of overlap, python uses a well-defined “method resolution order” to decide which to use
  + Method resolution order and super()
  + Base class initialization
    - If a class uses multiple inheritance and defines no initializer, only the initializer of the first base class is automatically called
  + \_\_bases\_\_ of class objects
    - Tuple of class’s base classes
  + Ex)
    - SortedIntList.\_\_bases\_\_
* Method Resolution Order
  + Method Resolution Order(MRO)
    - Ordering of an inheritance graph that determines which implementation to use when invoking a method
    - Method implementation may be found in any class in an inheritance graph
    - Determines the order in which the graph is searched when looking for an implementation
  + \_\_mro\_\_ to access tuple with classes defining resolution order
  + How is MRO used
    - Python finds the MRO for the type of the object on which a method is invoked
    - Python checks each class in the MRO in order to find one that implements the method
    - The first implementation found is used
  + object
    - the ultimate base class for every class in Python
  + C3
    - Algorithm used to calculate method resolution orders
    - Ensures that subclasses come before base classes
    - Base class order from class definition is preserved
    - The first two qualities are preserved for all MROs in a program
    - Prohibits some inheritance declarations in python
* super()
  + given a method resolution order and a class C in that MRO, super() gives you an object which resolves methods using only the part of the MRO which comes after C
  + super() works with the MRO of an object, not just its base classes
  + super() gives you a proxy object
  + the proxy resolves the correct implementation if any requested method
  + super() have access to the entire inheritance graph of the object
  + proxy gets two important arguments
    - invoking class
    - object to resolve MRO
  + invoking class must b e a member of MRO used
  + the proxy resolves \_\_init\_\_ using classes in MRO after class
  + MRO uses self of the type
  + Example
    - class SortedIntList(IntList, SortedList):
    - SortedIntList.\_\_mro\_\_ returns (<class ‘simple\_list.SortedIntList>, <class ‘simple\_list.IntList>, ,<class ‘simple\_list.SortedList>, <class ‘simple\_list.SimpleList’>, <class ‘object’>
    - Proxy resolves using class listed aft ..IntList for name resolution
    - SortedList, SimpleList and object
  + Super() uses the full MRO of an object, not just the base classes from a class definition
  + Instance bound super proxies
    - Get mro used for named resolutions by finding type of instance(self)
    - Generally a super() used in an instance method will be instance bound
  + Class-bound super proxies
    - Use super() in a class method
    - Have a class object to work with
    - super() derives the MRO from the class object rather than the type of self
  + class method example
    - class Animal:
    - @classmethod
    - def description(cls):
    - return ‘An animal”
  + To call class methods [class].[method]
  + Ex)
    - Animal.description()
  + Instance bound super()
    - Gets MRO by looking at the type of the first argument
  + Class bound super()
    - Gets MRO by looking at the argument itself
  + super() will behave in an intuitive way
  + Explicit Arguments to super()
    - super(class-object, instance-or-class)
    - class-object: where to trim MRO
    - instance-or-class: provides MRO
* Resolving the Mystery
  + super()
    - given a class
    - and an MRO the contains the class
    - super takes everything in mro after class and uses that as new mro for resolving methods
    - all bundle in proxy objects that is return from super() call
* Summary
  + In python, the type of an object doesn’t determine if it can be used in a particular context
  + Python uses duck typing where fitness for purpose is determined at the time of use
  + Functions don’t specify their types
  + You can call any method on any object, and python won’t complain until runtime
  + Nominally type languages
    - Type matching: compilers enforce that only objects of the correct type are passed to functions
    - Satisfying the type system can become very significant element in your development effort
  + Python
    - No need to satisfy a static type system
    - You don’t need to use inheritance in python to bestow objects with types
    - Inheritance in python is best used for sharing implementation
  + Use a comma-separated list of class names for multiple base classes
  + A class can have as many base classes as you want
  + You should generally explicitly initialize base classes
  + Python will call, at most, the initializer of the first of multiple base class
  + Python will call a base class initializer only if the subclass doesn’t define one
  + \_\_bases\_\_ is a tuple defining the base classes for the class
  + \_\_bases\_\_ is in the same order as in the class definition
  + \_\_bases\_\_ is populated for both single and multiple inheritance
  + Method resolution order is the order in which python searches an inheritance graph
  + MRO is a tuple of types in the \_\_mro\_\_ attribute
  + Python uses the first entry in an MRO which has the method
  + MRO is dependent on base class declaration order
  + MRO is calculated by python using the C3 algorithm
  + C3 preserves base-class declarations order
  + C3 puts subclasses before base classes
  + It is possible to specify an inconsistent base class ordering
  + super() uses the elements in an MRO after some specified type
  + super() returns a proxy object
  + a super proxy uses a subset of an MRO for name resolution
  + you can’t directly call instance method on class-bound proxies
  + Inappropriate use of super() can violate design constraints
  + Classes can be designed to cooperate without a priori knowledge of one another
  + Object is at the core of python’s object model
  + Object is the ultimate base class for all other classes
  + Python will automatically provide object as a base class
  + Object provides default implementations of many common python methods
  + Object implements the core attribute functionality in python
  + Inheritance in python is best used as a way to share implementation
* Overview
  + Programmatically transform class definitions
  + similar mechanism to function decorators
  + Metaprogramming - treating programs as data
  + Overlap with the capabilities of metaclasses
  + Less powerful than metaclasses but easier to use
  + Class decorators often introspect the decorated class
  + function object -> decorated by @decorator ->creates wrapper function object -> wraps function object -> rebind variable to wrapper function object
  + syntax
    - @decorator
    - class C:
  + common alternative for class decorator
    - modify decorated object in place
    - rather than wrap it and return the wrapper
    - classes provide more opportunity for in place modification
  + class object -> decorated by @ decorator -> modifies class object -> rebinds variable to class object
* Can We Synthesize a Method?
* Making a Class Decorator
  + identity function: a function that returns its argument
  + ex)
    - def auto\_repr(cls):
    - return cls
    - @auto\_repr
    - class Location:
    - ….
  + vars(cls)
    - returns a mapping from member name to member objects
  + class declaration are applied when decorated class is first being defined
    - when it is imported
  + modules are singleton
  + inspect.signature(class\_\_init\_\_)
    - returns signature object that contains argument list for \_\_init\_\_
* Class Decorator Factories
  + def invariant(predicate):
  + function\_decorator = postcondition(predicate)
  + def class\_decorator(cls):
  + members = list(vars(cls).items())
  + for name, member in members:
  + if inspect.isfunction(member):
  + decorated\_member = function\_decorator(member)  
     setattr(cls, name, decorated\_member)
  + return cls
  + return class\_decorator
* Summary
  + class decorators transform class definitions
  + class decorators are unary function which accept a class object, cls
  + class decorator should return a class object, often the same on they accept
  + class decorators are a simpler alternative to metaclasses
  + class decorator factories facilitate parameterization
  + multiple class decorators can be applied
* Data Classes
  + the data class concept
* Defining Data Classes
  + from dataclasses import dataclass
  + @dataclass
  + class Location:
  + name: str
  + position: Position
  + str and Position are type annotations
  + @dataclass decorator
    - Collect class attributes and use them to synthesize implementations of \_\_init\_\_ which accepts arguments with these names
    - And \_\_repr\_\_
  + Ex instantiation
    - paris = Location(“Paris”, Position(48.8, 2.3))
  + additional controls over which methods are generated are provided via arguments pass to decorator
  + ex)
    - @dataclass(
    - init=True, #enable \_\_init\_\_
    - repr=True, #enable \_\_repr\_\_
    - eq=True, #enable \_\_eq\_\_ for comparison
    - order=False, #enable \_\_lt\_\_, \_\_gt\_\_, etc
    - unsafe\_hash=False,
    - frozen=False
    - )
* Hash and Hashability
  + Hash-based collections require immutable elements
  + Equality and hashing must be consistent
  + Dataclasses are best used to represent immutable value objects
    - Use immutable attribute types
    - Declare the dataclass as frozen(immutable)
  + Ex)
    - @dataclass(eq=True, frozen=True)
* Dataclass Invariants
  + Encapsulation
    - Managed access to hidden data
  + Abstraction
    - Simple interfaces to complex objects
  + Inheritance
    - Relating the general to the specific
  + Polymorphism
    - A single interface to different types