ITP 115 – Programming in Python

Inheritance and Polymorphism

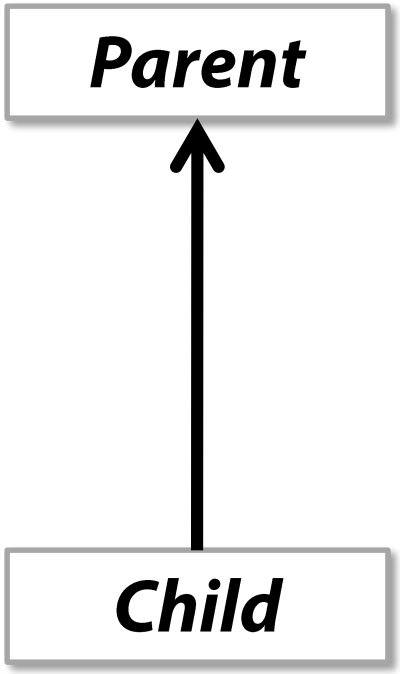
Objects

- Objects are a great tool to reduce errors and provide code organization
- We spend a great deal of time designing attributes and methods
- …Wouldn't it be great if we could reuse our custom objects even more?

Inheritance

- Allows a programmer to define a general class
- Later can define a more specific class
  - Add more details to the general class
- New class ***inherits*** all the properties of the general class and gets all the details of the specific class

Inheritance Analogy

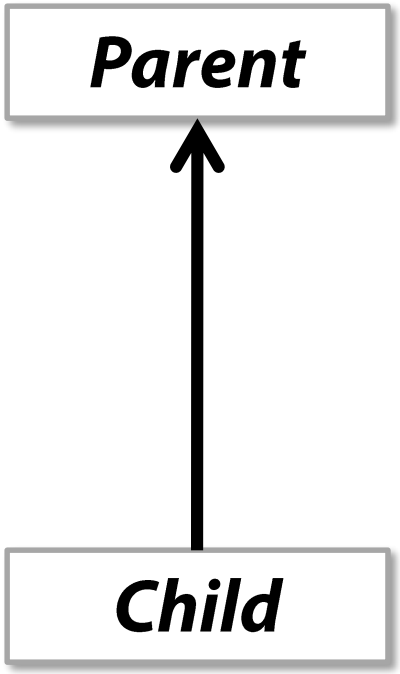


What is passed on to a child?

Traits (Attributes)

Behaviors (Methods)

Inheritance Analogy



What is passed on to a child?

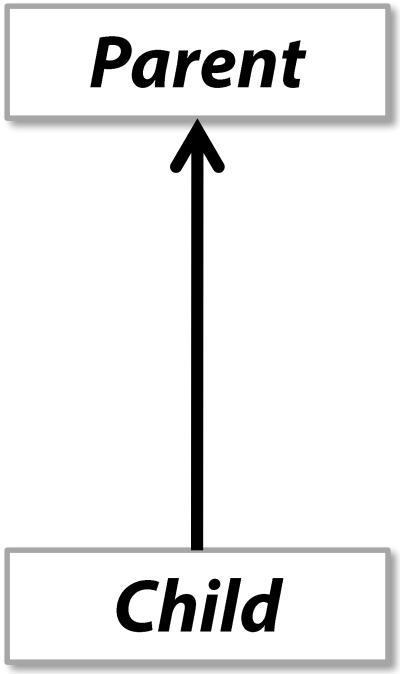
Traits (Attributes)

- *Eye Color*
- *Hair Color*

Behaviors (Methods)

- *How you cook*
- *How you manage money*

Inheritance Analogy



In Python, the same happens

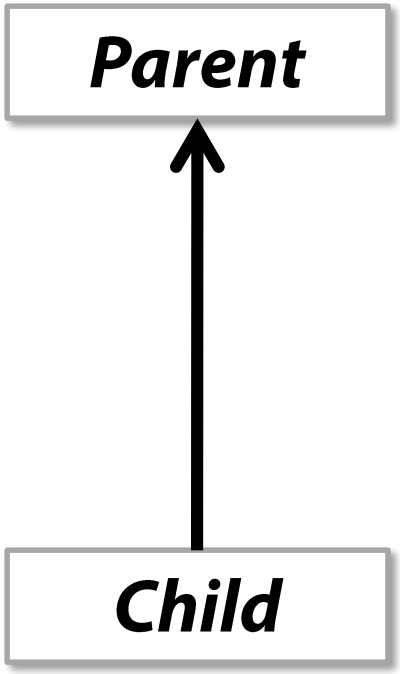
Traits (Attributes)

- *Data variables of a class*

Behaviors (Methods)

- *Methods of a class*

Inheritance Terms



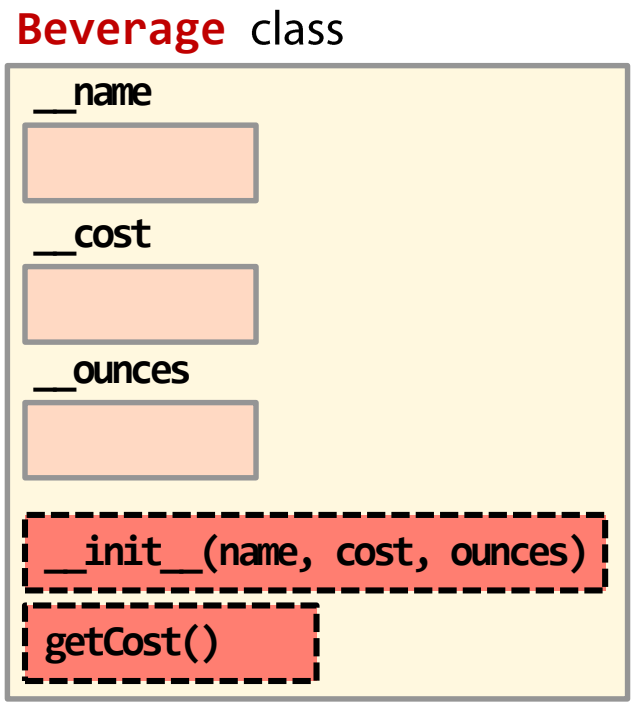
Terminology

Super Class, Parent Class, Base Class

Sub Class, Child Class, Derived Class

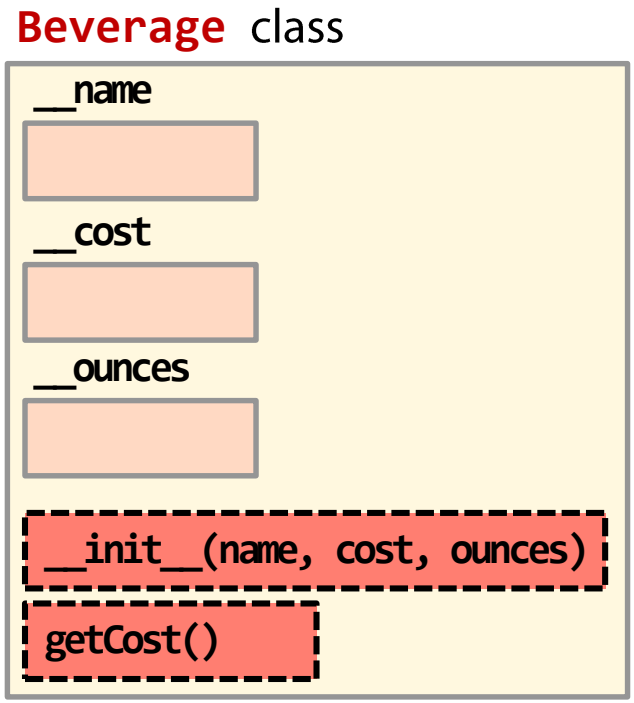
Example: Beverage class

- **Beverage**
  - Private attributes
  - Constructor
  - Getters/setters (not all shown)
  - Method



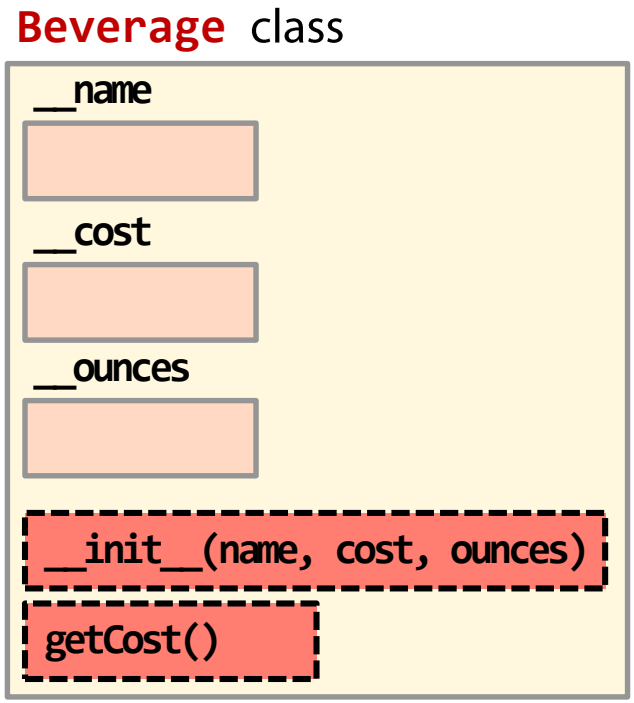
Example: Beverage class

- **Beverage** is very general
  - Let's make more specific types of drinks
  - Ex: soda, tea, coffee
- Let's create a **Tea** class



Example: Beverage class

- Rather than duplicating all the code (e.g. getters), **Tea** will inherit from **Beverage**
- **Tea** will have all the same attributes and methods as **Beverage**
- Consider what new attributes the **Tea** class will have that **Beverage** did not

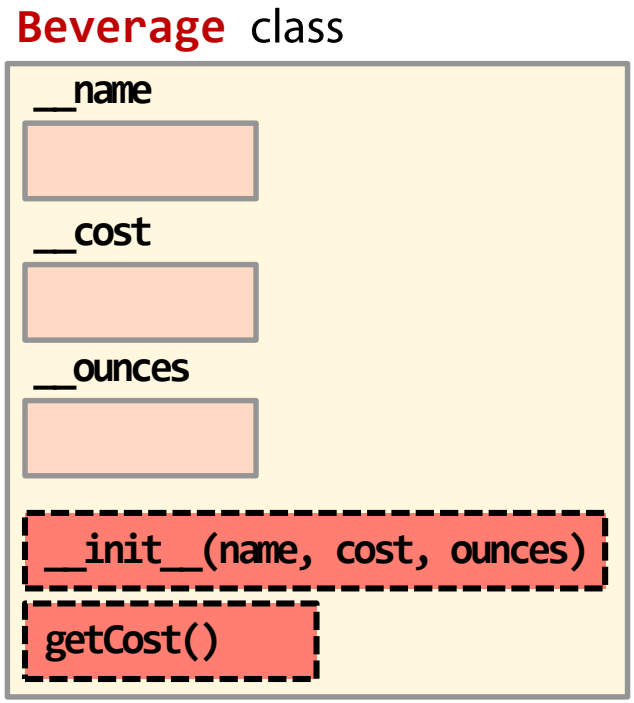


Example: Beverage class

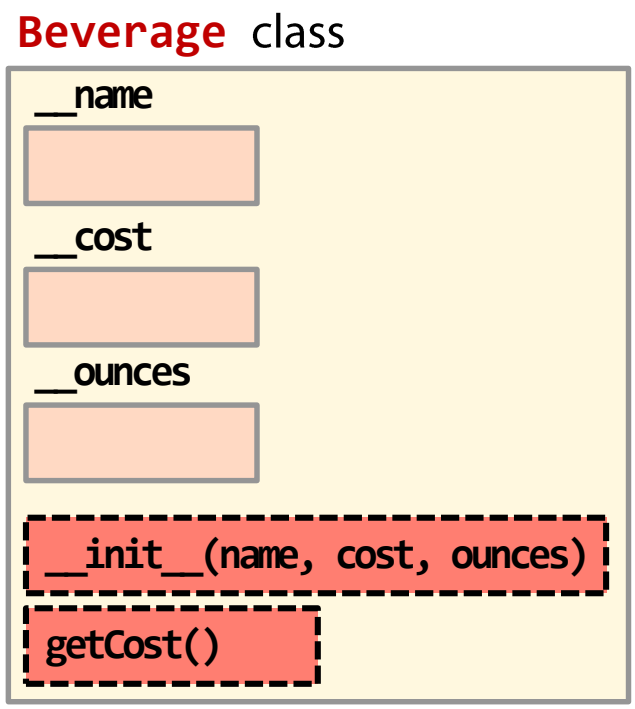
- Let's say the **Tea** class will have a new attribute called **___caffeinePerOunce**

- Does this affect the constructor?

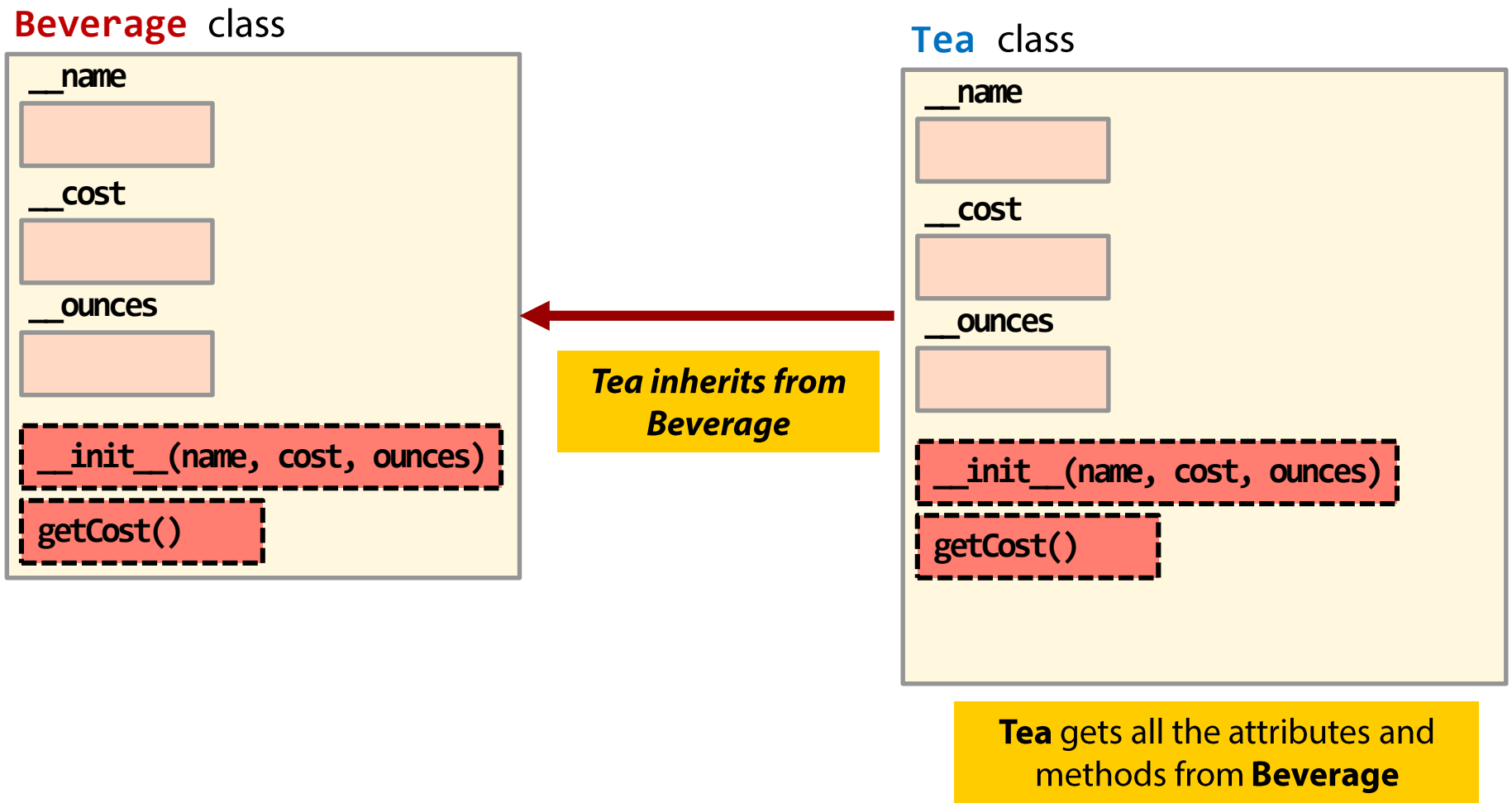
- Does this affect **getCost()** ?



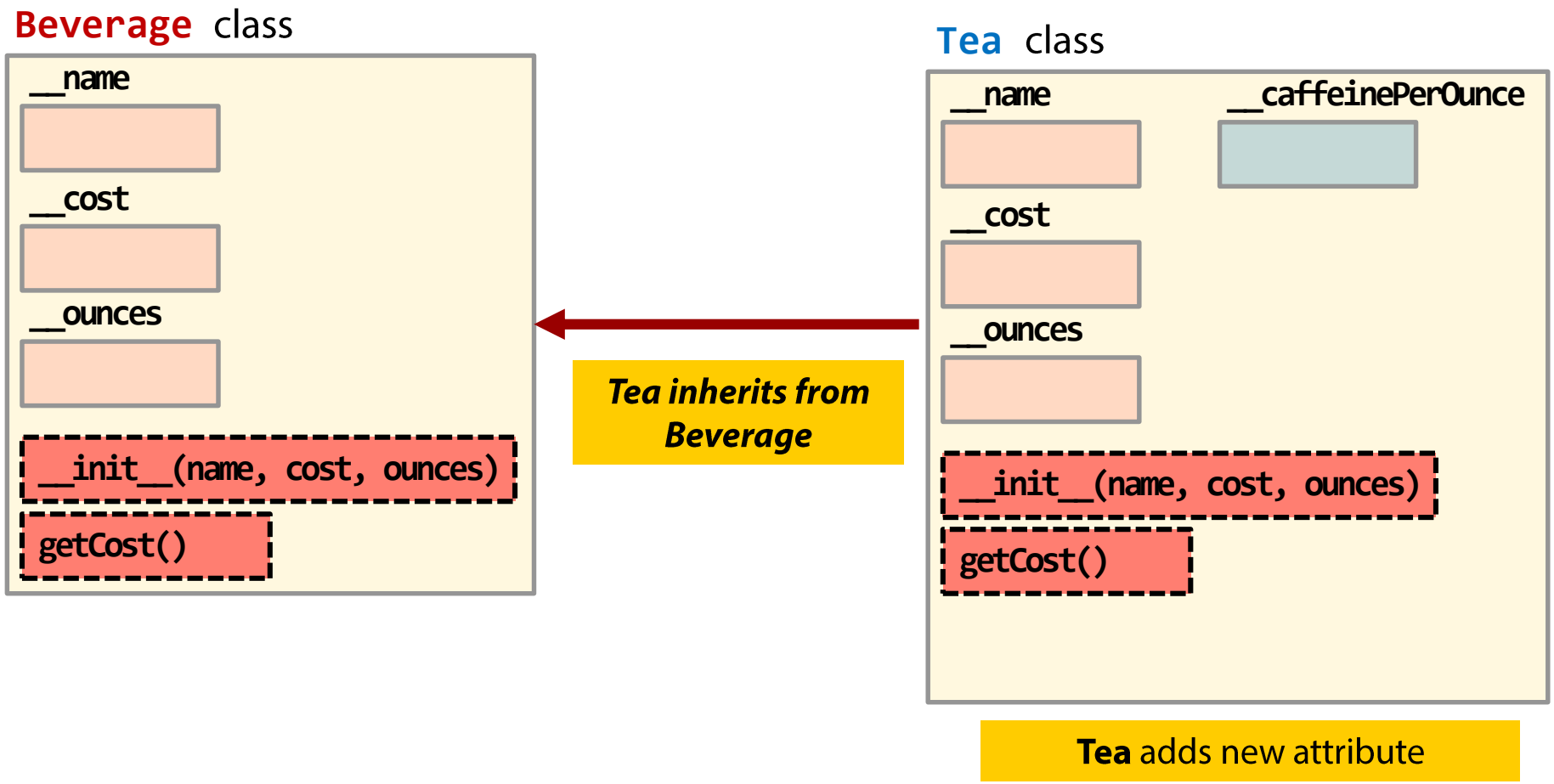
Example: Beverage class



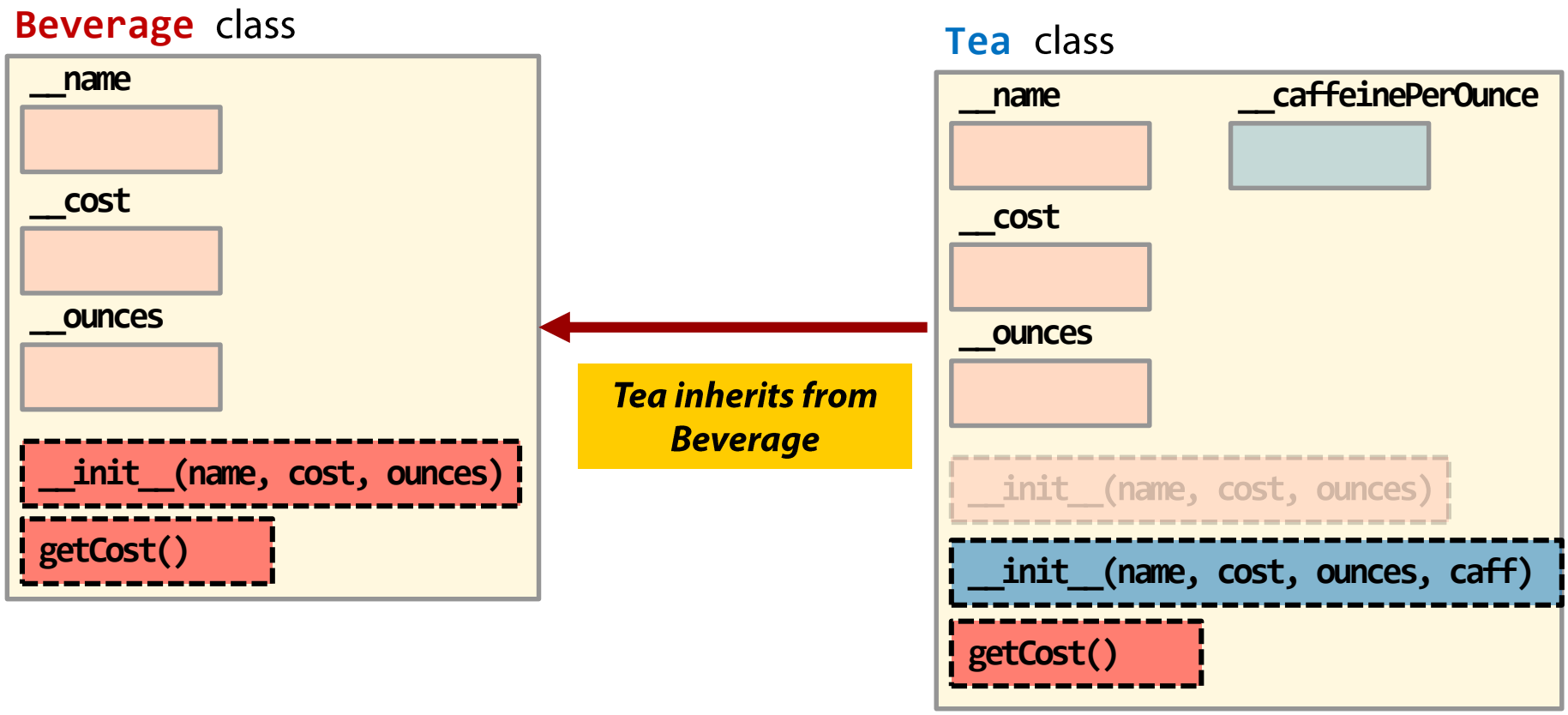
Example: Beverage class



Example: Beverage class

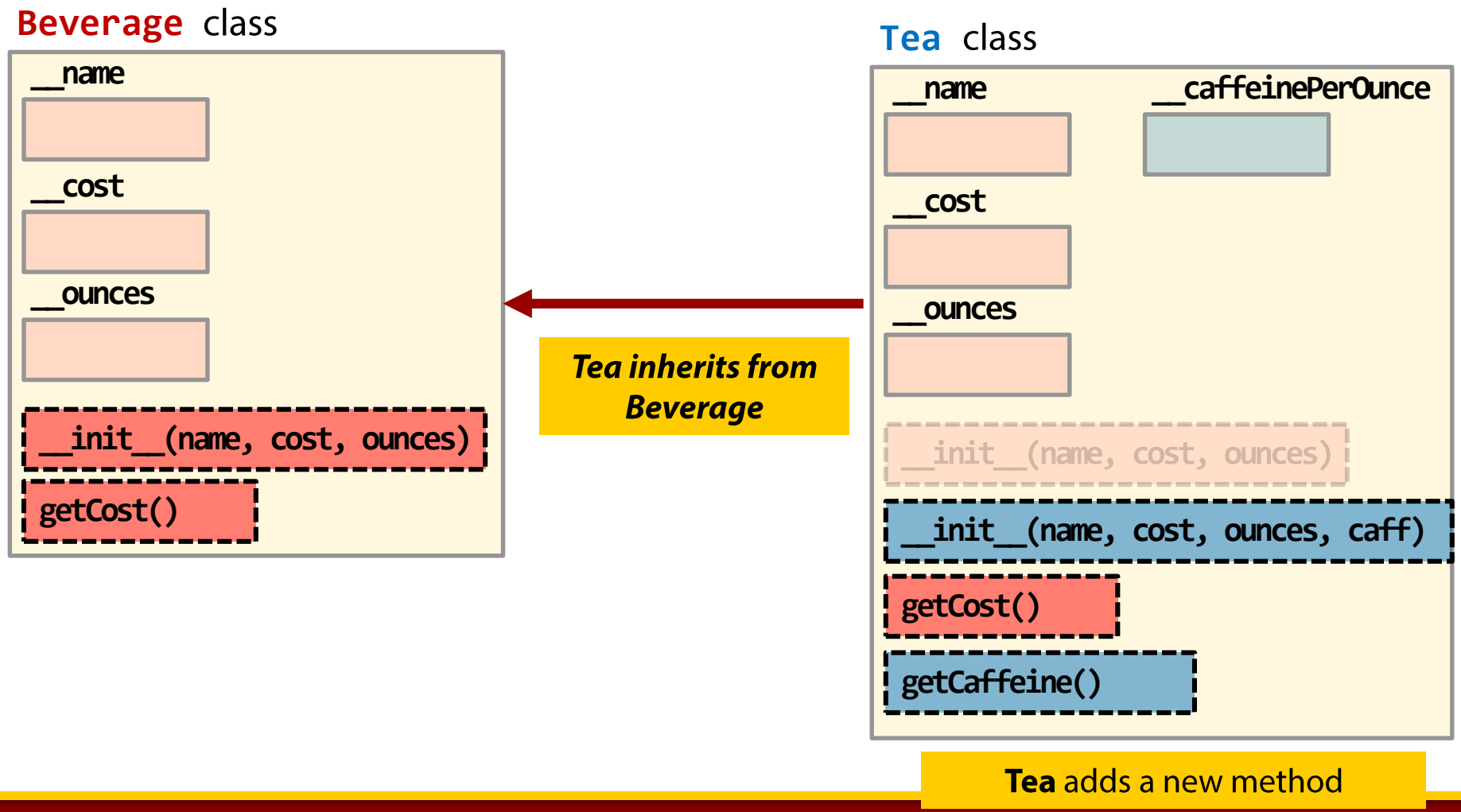


Example: Beverage class



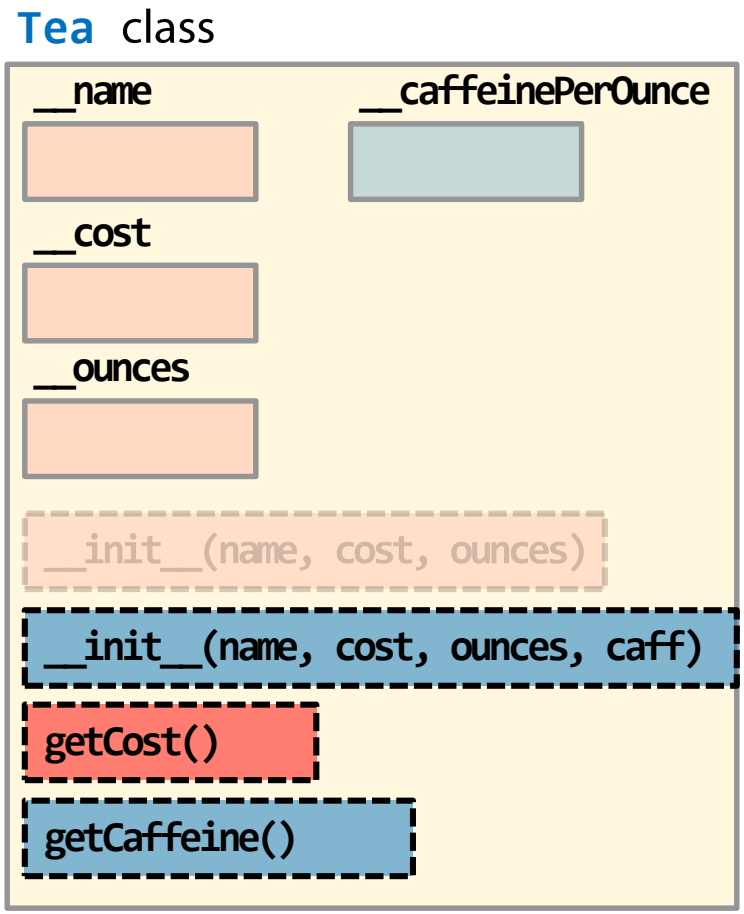
Tea overrides the parent constructor and makes a new one

Example: Beverage class

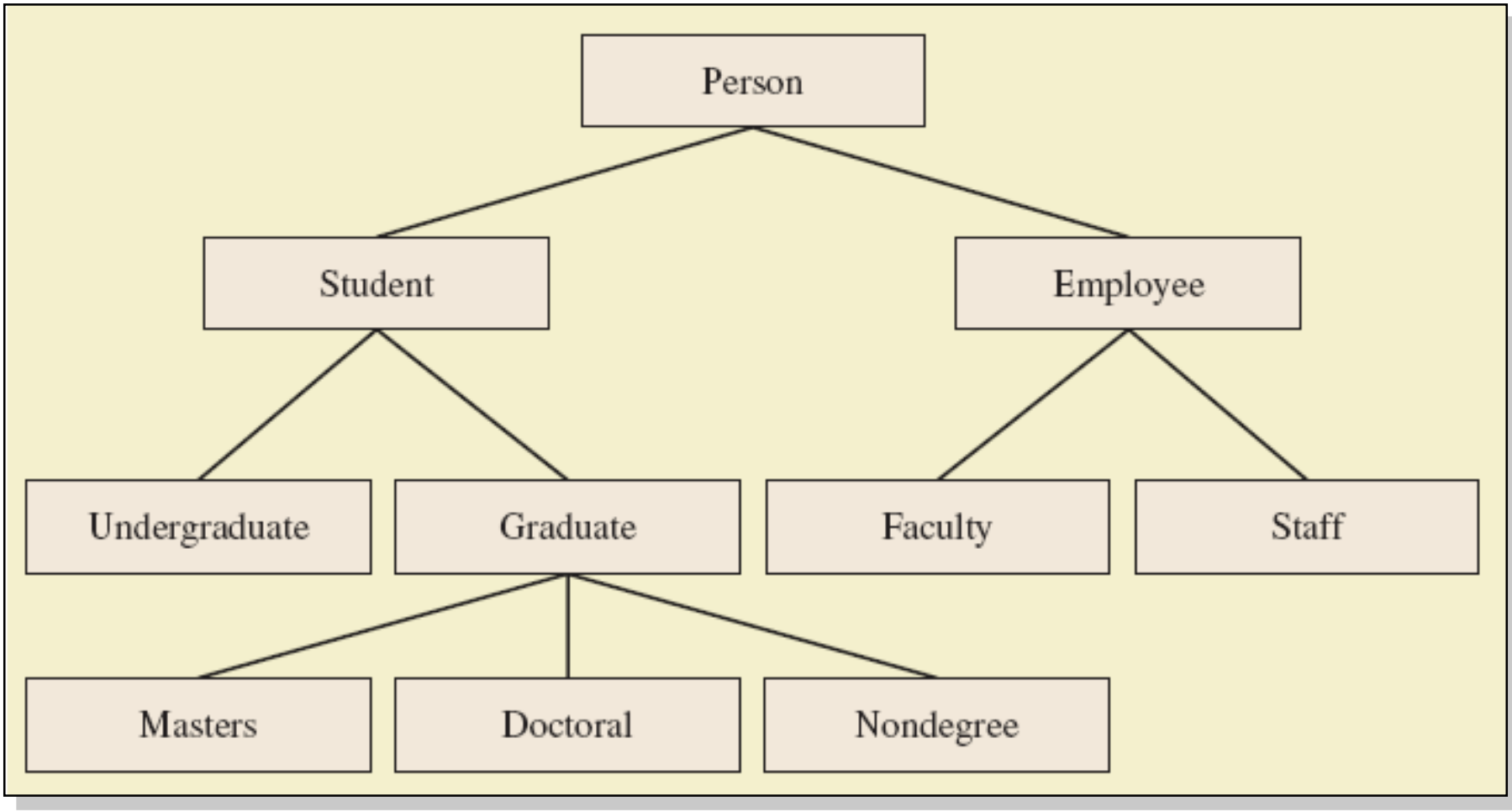


Example: Beverage class

- What does this all mean?
- The **Tea** class inherits all the variables and methods of the **Beverage** Class
  - Ex: `__cost`
- The **Tea** class also creates new variables and methods
  - Ex: `__caffeine`



Hierarchy of Inheritance



Inheritance Advantages

- Easier to code
- Less errors
- Easier to understand

INHERITANCE IN PYTHON

Creating Inheritance

- Consider two classes: **Beverage** and **Tea**

```
class Beverage(object):
```

- To establish inheritance from **Beverage** (parent) to **Tea** (child)

```
class Tea(Beverage):
```

Constructors and Inheritance

- In the child (`Tea`) constructor, the first thing you must do is call the parent constructor
- This is necessary to create the attributes that were defined in the parent (`Beverage`)
- After this, take care of any new attributes in the child (`Tea`) class

Constructors and Inheritance

```
class Beverage(object):
  def __init__(self, name, cost, ounces):
      self.__name = name
      self.__cost = cost
      self.__ounces = ounces
```

Beverage constructor takes 3 parameters and creates 3 attributes

Constructors and Inheritance

```
class Tea(Beverage):
  def __init__(self, name, cost, ounces, caff):
```

Constructors and Inheritance

```
class Tea(Beverage):
  def __init__(self, name, cost, ounces, caff):
```

These parameters will go to parent constructors

This parameter is for new our attribute

Constructors and Inheritance

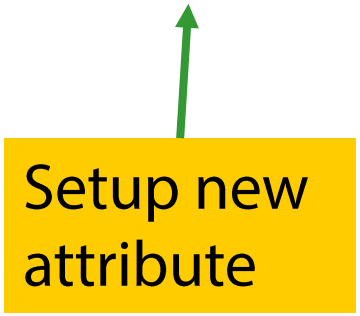
```
class Tea(Beverage):
    def __init__(self, name, cost, ounces, caff):
        super().__init__(name, cost, ounces)
```

super() always refers to the parent class

super().__init__ Explicitly calls the constructor in the parent class

Constructors and Inheritance

```
class Tea(Beverage):
  def __init__(self, name, cost, ounces, caff):
      super().__init__(name, cost, ounces)
      self.__caffeinePerOunce = caff
```



Defining New Child Methods

- It is easy to define new methods
  - Meaning methods that didn't original exist in parent
- Simply define methods as you would in any class

Defining New Child Methods

```
class Tea(Beverage):
  ...

  def setCaffeine(self, caff):
      self.__caffeinePerOunce = caff

  def getCaffeine(self):
      return self.__caffeinePerOunce
```

Inherited Methods

- For each method a child class inherits, we can one of the following:
  1. Do nothing and inherit the methods
  2. Override the method
     - Complete redefine the method does
  3. Override the method BUT also call the parent version

Inheriting Methods

```
class Tea(Beverage):
  ...
  # do nothing with getCost()

def main():
  t = Tea("White Tea", 2, 8, 30)
  print(t.getName())
```

```
White Tea
```

- Shape exercise (version 1)

Overriding Methods

- Let's say the child class needs to **redefine** what a an inherited method does
  - Ex: **__str__**

- *Overriding* a method **blocks** the parent version and replaces it with the new version

- To override method, define a method with the same name and parameters
  - Should also have some return type

Overriding Methods

```
class Beverage(object):
   def __str__(self):
       msg = self.__name + ", "
       msg += "$" + str(self.__cost) + ", "
       msg += str(self.__ounces) + " ounces"
       return msg

def main():
   b = Beverage("Water", 1.25, 8)
   print(b)
```

```
Water, $1.25, 8 ounces
```

Overriding Methods

```
class Tea(Beverage):
   def __str__(self):
       msg = self.__name + ", "
       msg += "$" + str(self.__cost) + ", "
       msg += str(self.__ounces) + " ounces, "
       msg += str(self.__caffeinePerOunce) + " mg caff"
       return msg

def main():
   t = Tea("White Tea", 2, 8, 30)
   print(t)
```

```
White Tea, $2, 8 ounces, 30 mg caff
```


*This is just for illustration. We will see shortly this wouldn't work

Overriding Methods

- Overriding / redefining parent method can be very useful

- What if we want the best of both approaches?
  - We want the original operations from the parent method
  - BUT we also want to add to the new method

Calling an Overridden Method

- The keyword **`super`** can also be used to call an the original parent method from inside the child method

- Think back to **__str__**
  - We needed to redefine **__str__** but had a lot of duplicate code

Overriding Methods

```
class Tea(Beverage):
   def __str__(self):
      msg = self.__name + ", "
      msg += "$" + str(self.__cost) + ", "
      msg += str(self.__ounces) + " ounces"
      msg += str(self.__caffeinePerOunce) + " mg caff"
```

Had already been defined in parent method **__str__**

Overriding Methods

```
class Tea(Beverage):
   def __str__(self):
      msg = super().__str__()
      msg += str(self.__caffeinePerOunce) + " mg caff"
```

Replace with explicit call to parent method

Overriding Methods

```
class Tea(Beverage):
   def __str__(self):
       msg = super().__str__()


       msg += str(self.__caffeinePerOunce) + " mg caff"
```

Add any new operations

- Shape exercise (version 2)

Privacy and Inheritance

- What happens when an attribute is declared private in the parent class?

Privacy and Inheritance

```
class Tea(Beverage):
  def sip():
      print("Mmm! Sipping", self.__name)
```

Privacy and Inheritance

```
class Tea(Beverage):
  def sip():
      print("Mmm! Sipping", self.__name)

def main():
  t = Tea("White Tea", 2, 8, 30)
  print(t)
```

Error!

Why?

Privacy and Inheritance

```
class Tea(Beverage):
  def sip():
      print("Mmm! Sipping", self.__name)
```

This attribute is private so it can only be **directly** access in the Beverage class

Privacy and Inheritance

```
class Tea(Beverage):
  def sip():
      print("Mmm! Sipping", self.getName())
```

Use getter instead

Privacy and Inheritance

- Private attributes can only be **directly** accessed from within the parent class

- Child classes need to use **getters** and **setters**

Composition

- A class can contain (as an attribute) an object of another type

```
class Beverage(object):
    def __init__(self, name, cost, ounces):
        self.__name = name
```

- We say the **Beverage** **"has a"** string attribute called **__name**

Inheritance

- Questions:
- Is every `Tea` a `Beverage?`
- Is every `Beverage` a `Tea?`

Inheritance

- An child class can be used anywhere its parent is expected

```
class Tea(Beverage):
```

- We say the `Tea` **"is a"** `Beverage`

- Note
  - Every `Tea` **"is a"** `Beverage`
  - Not true the other way

(End Lecture)

- Game exercise

The `object` class

- The class called **`object`** is the ultimate parent of every class

```
class Beverage (object):
```

- This class defines all the "special" methods we have used

```
__init__
__str__
```

The `object` class

- The **`object`** class other useful methods we can override in our own custom classes

- Ex: Consider **`a`** and **`b`** are **`Beverage`** objects

Method	Use	What to Return
`__eq__(self, obj)`	`a == b`	Return **`True`** or **`False`**
`__gt__(self, obj)`	`a > b`	Return **`True`** or **`False`**
`__lt__(self, obj)`	`a < b`	Return **`True`** or **`False`**
`__add__(self, obj)`	`a + b`	Return new **`Beverage`** object
`__mult__(self, obj)`	`a * b`	Return new **`Beverage`** object

Ex: Custom Equals Method

```
class Beverage(object):
    def __eq__(self, otherBev):
        if (self.__name == otherBev.__name)
                and (self.__cost == otherBev.__cost)
                and (self.__ounces == otherBev.__ounces):
            return True
        else:
            return False

def main():
    a = Beverage("Water", 1.25, 8)
    b = Beverage("Water", 1.25, 7)
    print(a == b)
```

False