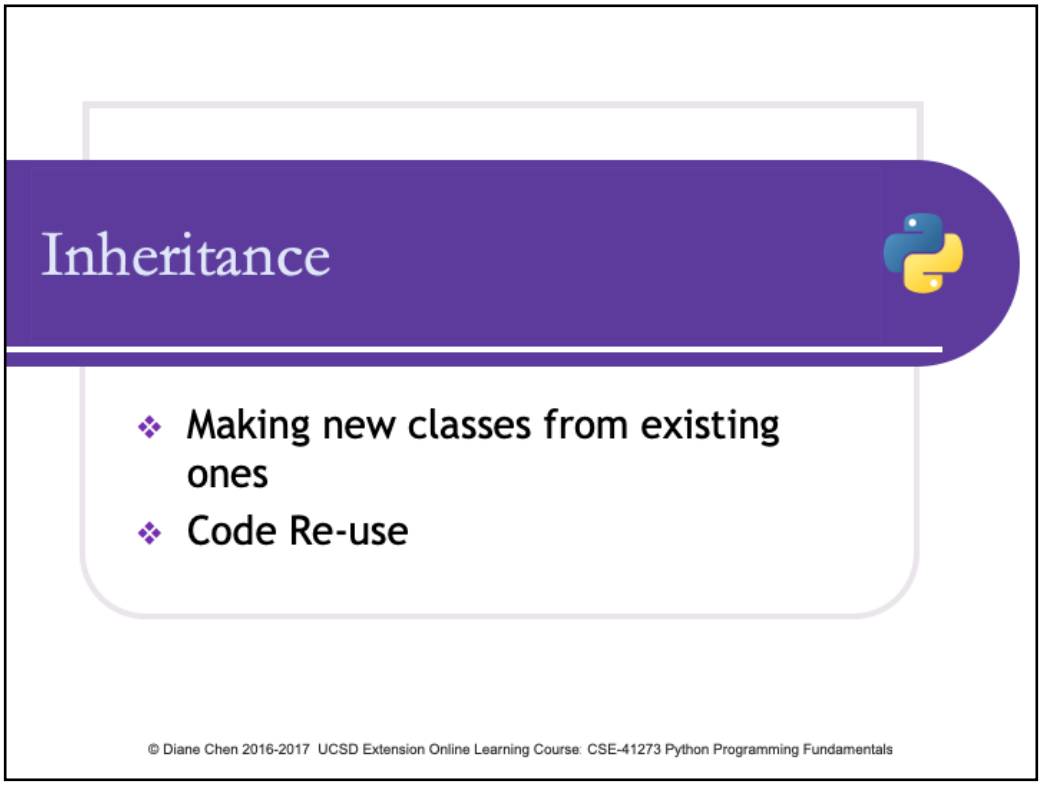
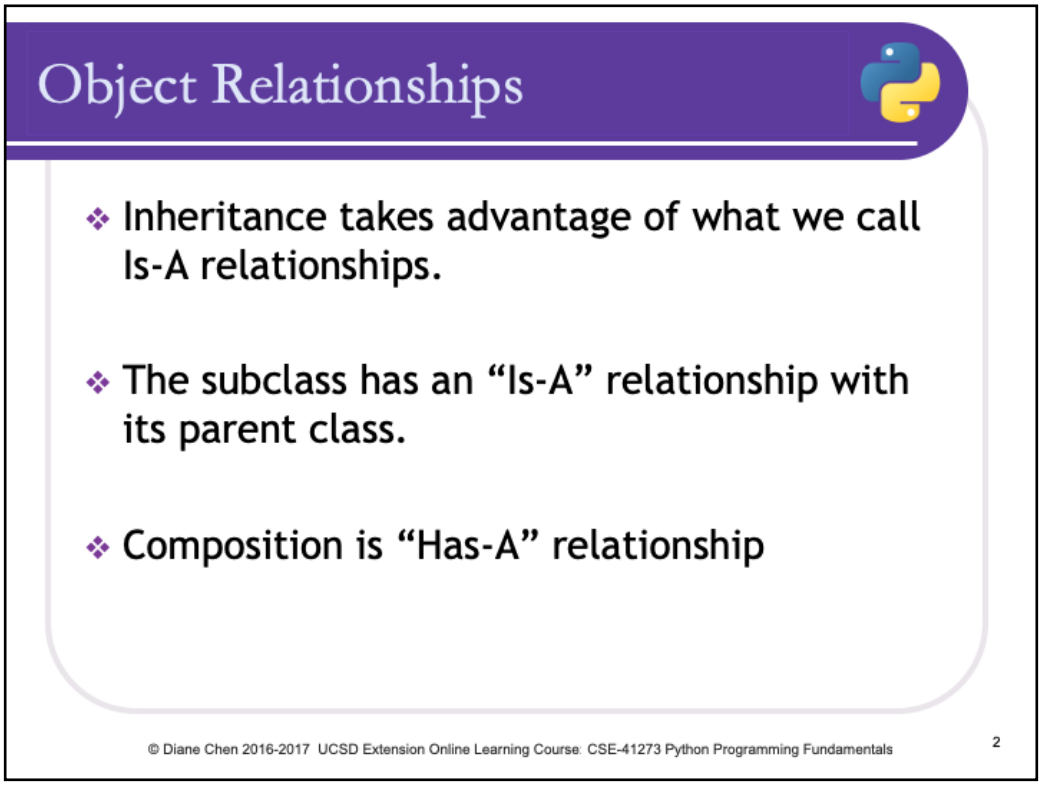
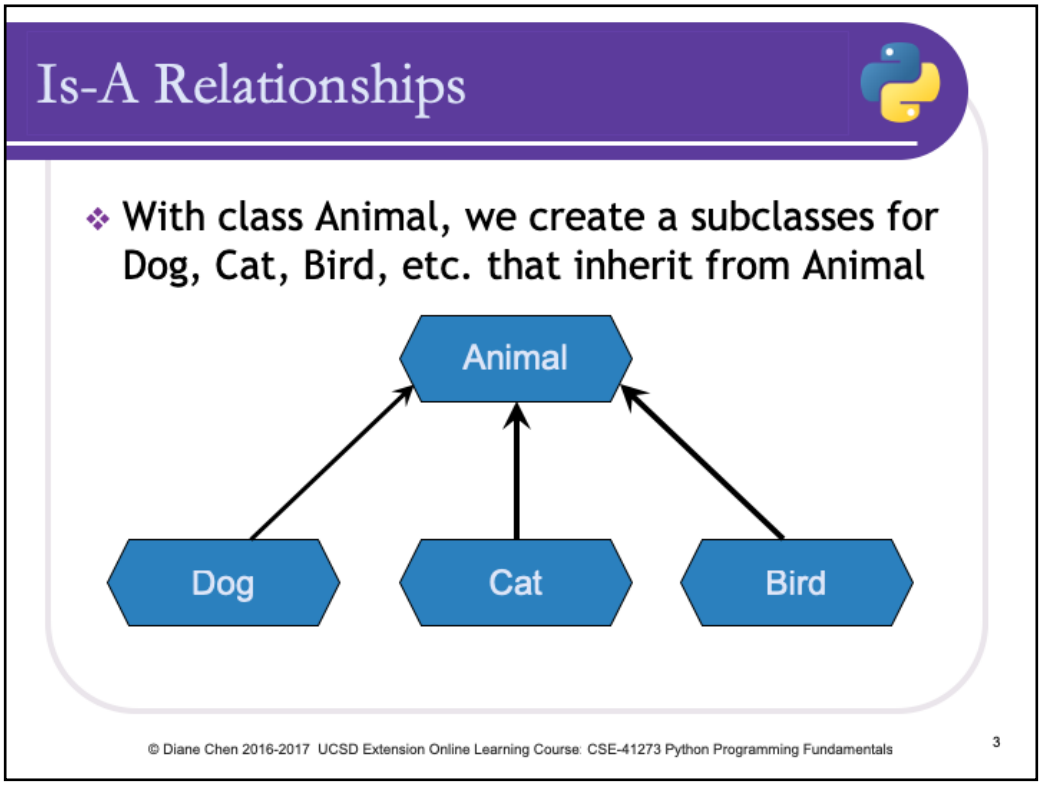


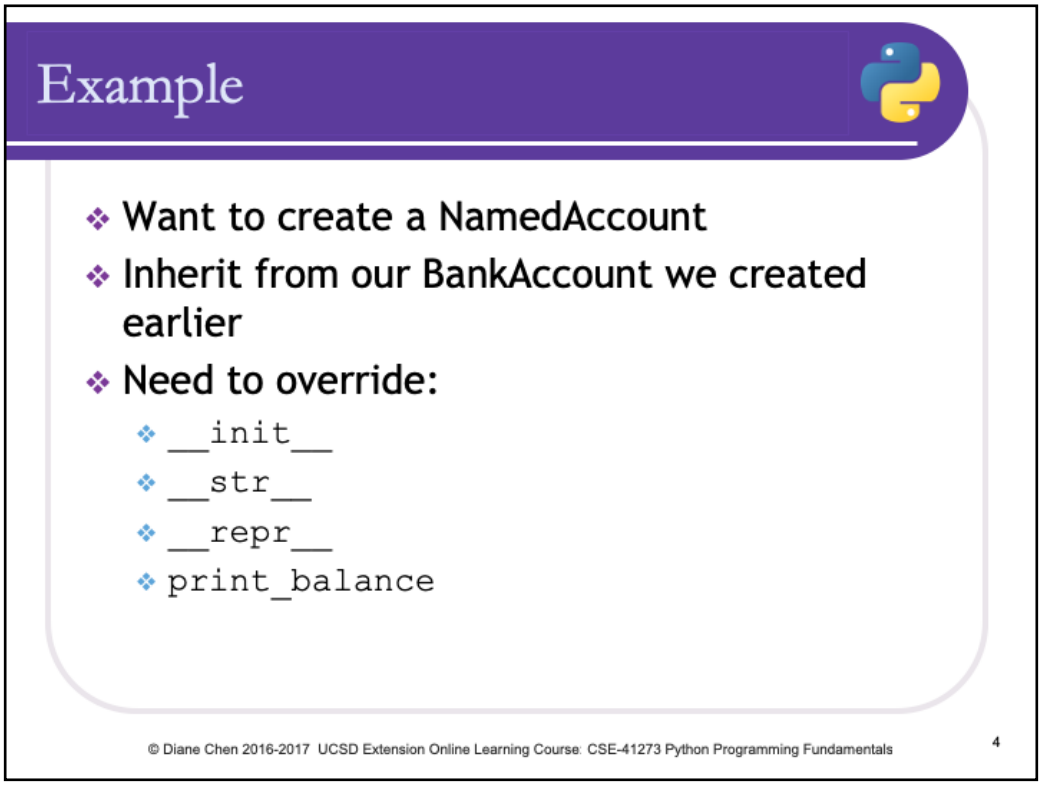
Now I'm going to talk about Inheritance for classes. In case I forgot to mention it before, one of the reasons for using classes is code re-use. With inheritance, we can make new classes from existing classes, giving us even more opportunity for code re-use. It should be the goal of every programmer to not have multiple versions of code for this or that reason. Classes aren't always the answer, but they can make life much easier by providing self-contained data and behavior that we can use without having to know the innards of the base class.



With object relationships, we want to define “Is-A” relationships for subclasses. A subclass should have an Is-A relationship of the parent class. In other words, if you have a class named Category and its subclass named Special, you should be able to say "An object of type Special Is-A Category". There is also composition, where the relationship between classes is that one class "Has-A" relationship with another class. This is different from inheritance, the Is-A relationship. With composition, the class with the Has-A relationship will be a part of or refernced by the other class. If you have class A that Has-A class B, then inside class A will be a reference to an object of class B. You will probably hear subclass, derived class, and child class terms used; they are the same thing. Likewise, you will hear base class, parent class, and superclass; they are also the same thing.



Say we have a class Animal, that contains information about animals; things that all animals have in common. Think about that for a minute. What kinds of information and behavior are shared among all animals? We can say that Dog has an Is-A relationship with Animal, namely, that a Dog is an Animal. The same for Cat and Bird. So we can create new classes for Dog, Cat, Bird that inherit from the Animal class. We say that Dog, Cat and Bird are subclasses of Animal and Animal is the parent class of Dog, Cat and Bird. The subclasses implement behavior and data unique to their class, but they can also share the data and behavior of the parent Animal class. Of course, this is a very simplistic example. It is just to show you how the Is-A relationship works.



Let's go through an example of making a subclass. Using the BankAccount class we created earlier, we would like to create a new NamedAccount class that stores names associated with the BankAccount. We can create a new class called NamedAccount, that inherits from BankAccount. What will we need to change and/or implement in our new class to get the behavior we want? We will need to override the dunder init method to handle getting the name input and saving it in the instance. Then we will want to override dunder str and dunder repr to print the name information with the account information. Then we will also want to override the print_balance method so that it will also print the name of the account. Our NamedAccount doesn't need any new methods that are unique to it; all we are doing is adding new data and adjusting way the account is printed.

```python
class BankAccount:
    def __init__(self, balance=0):
        self.balance = balance
        print("Account opened.")
        self.print_balance()
    def deposit(self, amount):
        self.balance += amount
        print("${} deposited.".format(amount))
        self.print_balance()
    def withdraw(self, amount):
        self.balance -= amount
        print("${} withdrawn.".format(amount))
        self.print_balance()
    def print_balance(self):
        print("Account balance is ${}.".format(self.balance))
    def transfer(self, other_account, amount):
        self.withdraw(amount)
        other_account.deposit(amount)
    def __str__(self):
        return "Account with balance of ${}".format(self.balance)
    def __repr__(self):
        return "BankAccount(balance={})".format(self.balance)
```


Here is the BankAccount class as we left it last week. It has all the methods we were using in our previous example using the BankAccount.

If you don't already have this file from the previous lesson, you can download "bank_account.py" from the link in this week's Notes and Resources page.

This is our new NamedAccount subclass. These lines of code go at the end of the bank_account.py file, but for ease of discussion, I have numbered the lines starting at 1. We could also put just this code into a file called "named_account.py" and add a line "from bank_account import BankAccount" spelled like the class name.

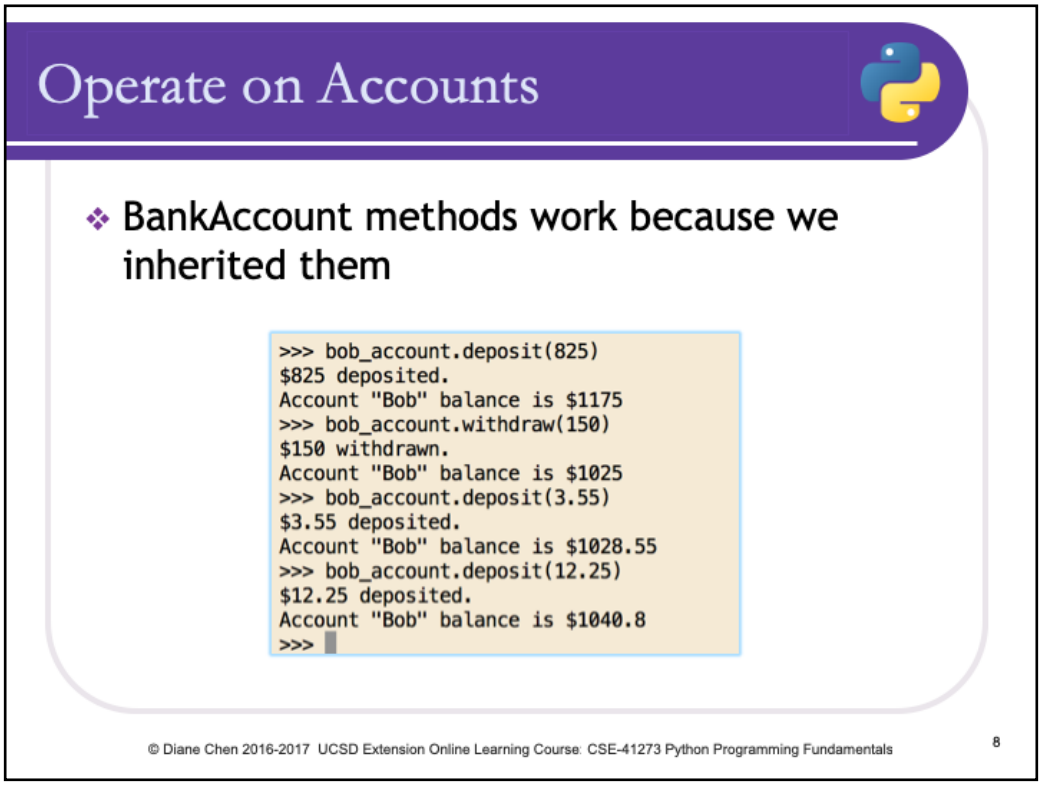
Take a look at what we changed in the dunder init method on lines 2 through 4. What is going on here? What we are doing is allowing any kind of inputs. The *args collects all the positional arguments and **kwargs collects all the keyword arguments. The only argument we care about is the name, so we remove it from the keyword argument list. On line 3, we “pop” the keyword argument "name” from the keyword arguments list and save it in our self object. “pop” has an optional argument of a default in case our desired element does not exist. So if no keyword argument "name" exists, the name is stored as None.

On line 4, we call super.dunder init, passing it all the remaining arguments. It looks pretty strange , but what this does is call the dunder init method for the superclass, giving it all the arguments that are left after we remove the “name” argument. Currently there is only one other argument to the BankAccount class, but by using the "*args" and "**kwargs", we make sure we pass all arguments to the parent class, and protect our NamedAccount from any changes that might occur to the BankAccount class. BankAccount can change its initialization and it won’t cause a problem for us.

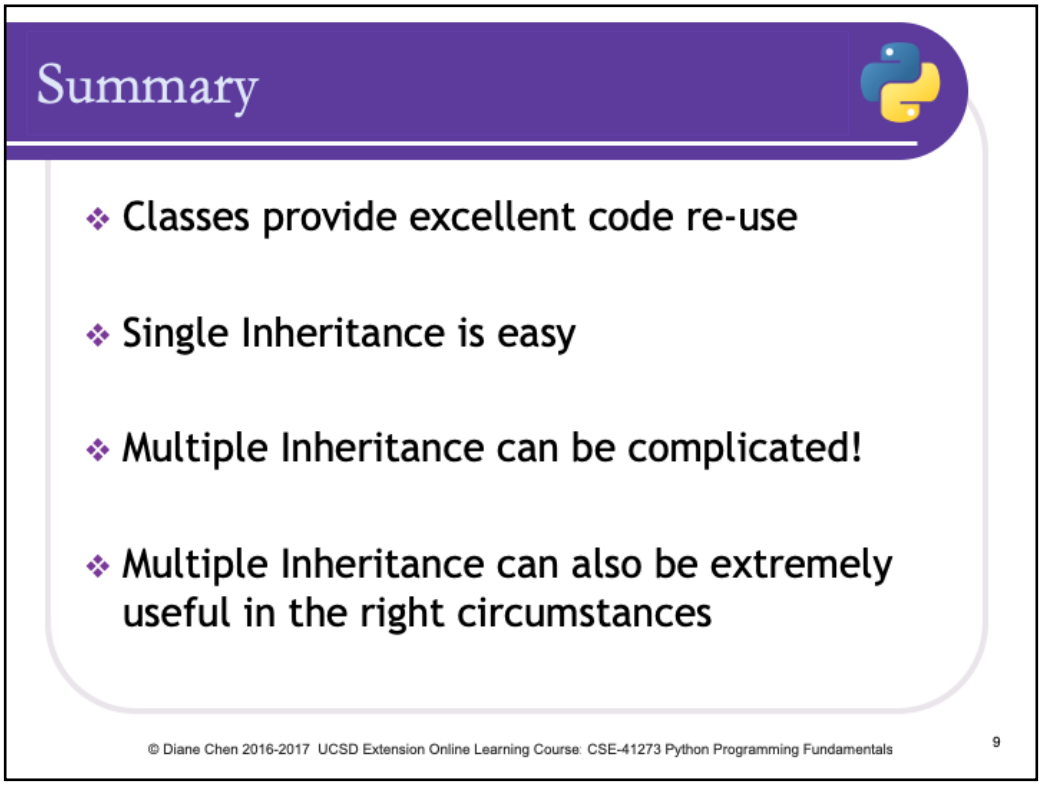
Create Accounts

```
>>> from bank_account import NamedAccount
>>> bob_account = NamedAccount(name="Bob", balance=350)
Account opened.
Account "Bob" balance is $350
>>> bob_account.name
'Bob'
>>> bob_account
NamedAccount(balance=350, name=Bob)
>>>
>>> nameless_account = NamedAccount()
Account opened.
Account "None" balance is $0
>>> nameless_account.name
>>> nameless_account.name is None
True
>>> nameless_account
NamedAccount(balance=0, name=None)
>>>
```


Here we see the creation of two accounts using the NamedAccount class. The first one, bob_account, is created with a name and a balance. It prints the account is opened, and print_balance prints the balance with the name Bob, because we are using our own version of print_balance(). We can access bob_account.name, since we have added the name as a property to the account. Typing bob_account prints the dunder repr string of the account information. Then we create a NamedAccount with no parameters, so we call it nameless_account. Because we didn't input a name, the name is stored as None. Note that when we just type nameless_account.name, nothing is printed, because the name is None and str() does not print anything for "None". So we can check if nameless_account.name is None, and the answer is True. When we type nameless_account, the dunder repr string is displayed, where there is a representation for the value None.



Because the NamedAccount is a subclass of the BankAccount class, we can apply all of the methods of the BankAccount class to our Namedaccount objects. We can deposit and withdraw to and from the account because they are methods of the base class.



So in summary, I'd like to reiterate that classes provide excellent opportunities for code re-use. When we are doing inheritance, single inheritance is pretty easy. Multiple inheritance can be very complicated and I don't recommend you try to do it yourself unless you have had lots of experience and understand the perils and pitfalls. I would say, for most programmers, you probably will only use multiple inheritance, if at all, for what we call "mixins", where we want to inherit from a main base class, then also inherit certain behaviors from some other classes that exist for this purpose.