Case study

Let's try to tie everything we've learned together with a larger example. We'll be designing a simple real estate application that allows an agent to manage properties available for purchase or rent. There will be two types of properties: apartments and houses. The agent needs to be able to enter a few relevant details about new properties, list all currently available properties, and mark a property as sold or rented. For brevity, we won't worry about editing property details or reactivating a property after it is sold.

The project will allow the agent to interact with the objects using the Python interpreter prompt. In this world of graphical user interfaces and web applications, you might be wondering why we're creating such old-fashioned looking programs. Simply put, both windowed programs and web applications require a lot of overhead knowledge and boilerplate code to make them do what is required. If we were developing software using either of these paradigms, we'd get so lost in GUI programming or web programming that we'd lose sight of the object-oriented principles we're trying to master.

Luckily, most GUI and web frameworks utilize an object-oriented approach, and the principles we're studying now will help in understanding those systems in the future. We'll discuss them both briefly in Chapter 13, *Concurrency*, but complete details are far beyond the scope of a single book.

Looking at our requirements, it seems like there are quite a few nouns that might represent classes of objects in our system. Clearly, we'll need to represent a property. Houses and apartments may need separate classes. Rentals and purchases also seem to require separate representation. Since we're focusing on inheritance right now, we'll be looking at ways to share behavior using inheritance or multiple inheritance.

`House` and `Apartment` are both types of properties, so `Property` can be a superclass of those two classes. `Rental` and `Purchase` will need some extra thought; if we use inheritance, we'll need to have separate classes, for example, for `HouseRental` and `HousePurchase` , and use multiple inheritance to combine them. This feels a little clunky compared to a composition or association-based design, but let's run with it and see what we come up with.

Now then, what attributes might be associated with a `Property` class? Regardless of whether it is an apartment or a house, most people will want to know the square footage, number of bedrooms, and number of bathrooms. (There are numerous other attributes that might be modeled, but we'll keep it simple for our prototype.)

If the property is a house, it will want to advertise the number of stories, whether it has a garage (attached, detached, or none), and whether the yard is fenced. An apartment will want to indicate if it has a balcony, and if the laundry is ensuite, coin, or off-site.

Both property types will require a method to display the characteristics of that property. At the moment, no other behaviors are apparent.

Rental properties will need to store the rent per month, whether the property is furnished, and whether utilities are included, and if not, what they are estimated to be. Properties for purchase will need to store the purchase price and estimated annual property taxes. For our application, we'll only need to display this data, so we can get away with just adding a `display()` method similar to that used in the other classes.

Finally, we'll need an `Agent` object that holds a list of all properties, displays those properties, and allows us to create new ones. Creating properties will entail prompting the user for the relevant details for each property type. This could be done in the `Agent` object, but then `Agent` would need to know a lot of information about the types of properties. This is not taking advantage of polymorphism. Another alternative would be to put the prompts in the initializer or even a constructor for each class, but this would not allow the classes to be applied in a GUI or web application in the future. A better idea is to create a static method that does the prompting and returns a dictionary of the prompted parameters. Then, all the `Agent` has to do is prompt the user for the type of property and payment method, and ask the correct class to instantiate itself.

That's a lot of designing! The following class diagram may communicate our design decisions a little more clearly:

Case study

Wow, that's a lot of inheritance arrows! I don't think it would be possible to add another level of inheritance without crossing arrows. Multiple inheritance is a messy business, even at the design stage.

The trickiest aspects of these classes is going to be ensuring superclass methods get called in the inheritance hierarchy. Let's start with the `Property` implementation:

```python
class Property:
    def __init__(self, square_feet='', beds='',
            baths='', **kwargs):
        super().__init__(**kwargs)
        self.square_feet = square_feet
        self.num_bedrooms = beds
        self.num_baths = baths

    def display(self):
        print("PROPERTY DETAILS")
        print("================")
```

```
            print("square footage: {}".format(self.square_feet))
            print("bedrooms: {}".format(self.num_bedrooms))
            print("bathrooms: {}".format(self.num_baths))
            print()

        def prompt_init():
            return dict(square_feet=input("Enter the square feet: "),
                    beds=input("Enter number of bedrooms: "),
                    baths=input("Enter number of baths: "))
        prompt_init = staticmethod(prompt_init)
```

This class is pretty straightforward. We've already added the extra `**kwargs` parameter to `__init__` because we know it's going to be used in a multiple inheritance situation. We've also included a call to `super().__init__` in case we are not the last call in the multiple inheritance chain. In this case, we're *consuming* the keyword arguments because we know they won't be needed at other levels of the inheritance hierarchy.

We see something new in the `prompt_init` method. This method is made into a static method immediately after it is initially created. Static methods are associated only with a class (something like class variables), rather than a specific object instance. Hence, they have no `self` argument. Because of this, the `super` keyword won't work (there is no parent object, only a parent class), so we simply call the static method on the parent class directly. This method uses the Python `dict` constructor to create a dictionary of values that can be passed into `__init__` . The value for each key is prompted with a call to `input` .

The `Apartment` class extends `Property` , and is similar in structure:

```
    class Apartment(Property):
        valid_laundries = ("coin", "ensuite", "none")
        valid_balconies = ("yes", "no", "solarium")

        def __init__(self, balcony='', laundry='', **kwargs):
            super().__init__(**kwargs)
            self.balcony = balcony
            self.laundry = laundry

        def display(self):
            super().display()
            print("APARTMENT DETAILS")
            print("laundry: %s" % self.laundry)
            print("has balcony: %s" % self.balcony)

        def prompt_init():
            parent_init = Property.prompt_init()
            laundry = ''
            while laundry.lower() not in \
                    Apartment.valid_laundries:
                laundry = input("What laundry facilities does "
                        "the property have? ({})".format(
                        ", ".join(Apartment.valid_laundries)))
            balcony = ''
            while balcony.lower() not in \
                    Apartment.valid_balconies:
                balcony = input(
                    "Does the property have a balcony? "
                    "({})".format(
                    ", ".join(Apartment.valid_balconies)))
            parent_init.update({
                "laundry": laundry,
                "balcony": balcony
            })
            return parent_init
        prompt_init = staticmethod(prompt_init)
```

The `display()` and `__init__()` methods call their respective parent class methods using `super()` to ensure the `Property` class is properly initialized.

The `prompt_init` static method is now getting dictionary values from the parent class, and then adding some additional values of its own. It calls the `dict.update` method to merge the new dictionary values into the first one. However, that `prompt_init` method is looking pretty ugly; it loops twice until the user enters a valid input using structurally similar code but different variables. It would be nice to extract this validation logic so we can maintain it in only one location; it will likely also be useful to later classes.

With all the talk on inheritance, we might think this is a good place to use a mixin. Instead, we have a chance to study a situation where inheritance is not the best solution. The method we want to create will be used in a static method. If we were to inherit from a class that provided validation functionality, the functionality would also have to be provided as a static method that did not access any instance variables on the class. If it doesn't access any instance variables, what's the point of making it a class at all? Why don't we just make this validation functionality a module-level function that accepts an input string and a list of valid answers, and leave it at that?

Let's explore what this validation function would look like:

```
def get_valid_input(input_string, valid_options):
    input_string += " ({}) ".format(", ".join(valid_options))
    response = input(input_string)
    while response.lower() not in valid_options:
        response = input(input_string)
    return response
```

We can test this function in the interpreter, independent of all the other classes we've been working on. This is a good sign, it means different pieces of our design are not tightly coupled to each other and can later be improved independently, without affecting other pieces of code.

```
>>> get_valid_input("what laundry?", ("coin", "ensuite", "none"))
what laundry? (coin, ensuite, none) hi
what laundry? (coin, ensuite, none) COIN
'COIN'
```

Now, let's quickly update our `Apartment.prompt_init` method to use this new function for validation:

```
    def prompt_init():
        parent_init = Property.prompt_init()
        laundry = get_valid_input(
                "What laundry facilities does "
                "the property have? ",
                Apartment.valid_laundries)
        balcony = get_valid_input(
            "Does the property have a balcony? ",
            Apartment.valid_balconies)
        parent_init.update({
            "laundry": laundry,
            "balcony": balcony
        })
        return parent_init
    prompt_init = staticmethod(prompt_init)
```

That's much easier to read (and maintain!) than our original version. Now we're ready to build the `House` class. This class has a parallel structure to `Apartment`, but refers to different prompts and variables:

```
class House(Property):
    valid_garage = ("attached", "detached", "none")
    valid_fenced = ("yes", "no")
    def __init__(self, num_stories='',
            garage='', fenced='', **kwargs):
        super().__init__(**kwargs)
        self.garage = garage
        self.fenced = fenced
        self.num_stories = num_stories

    def display(self):
        super().display()
        print("HOUSE DETAILS")
        print("# of stories: {}".format(self.num_stories))
        print("garage: {}".format(self.garage))
        print("fenced yard: {}".format(self.fenced))


    def prompt_init():
        parent_init = Property.prompt_init()
        fenced = get_valid_input("Is the yard fenced? ",
                    House.valid_fenced)
        garage = get_valid_input("Is there a garage? ",
                House.valid_garage)
        num_stories = input("How many stories? ")

        parent_init.update({
            "fenced": fenced,
            "garage": garage,
            "num_stories": num_stories
        })
        return parent_init
    prompt_init = staticmethod(prompt_init)
```

There's nothing new to explore here, so let's move on to the `Purchase` and `Rental` classes. In spite of having apparently different purposes, they are also similar in design to the ones we just discussed:

```
class Purchase:
    def __init__(self, price='', taxes='', **kwargs):
        super().__init__(**kwargs)
        self.price = price
        self.taxes = taxes

    def display(self):
        super().display()
        print("PURCHASE DETAILS")
        print("selling price: {}".format(self.price))
        print("estimated taxes: {}".format(self.taxes))

    def prompt_init():
        return dict(
            price=input("What is the selling price? "),
            taxes=input("What are the estimated taxes? "))
    prompt_init = staticmethod(prompt_init)

class Rental:
    def __init__(self, furnished='', utilities='',
            rent='', **kwargs):
        super().__init__(**kwargs)
        self.furnished = furnished
        self.rent = rent
        self.utilities = utilities

    def display(self):
        super().display()
        print("RENTAL DETAILS")
        print("rent: {}".format(self.rent))
        print("estimated utilities: {}".format(
            self.utilities))
        print("furnished: {}".format(self.furnished))

    def prompt_init():
        return dict(
            rent=input("What is the monthly rent? "),
            utilities=input(
                "What are the estimated utilities? "),
            furnished = get_valid_input(
                "Is the property furnished? ",
                    ("yes", "no")))
    prompt_init = staticmethod(prompt_init)
```

These two classes don't have a superclass (other than `object`), but we still call `super().__init__` because they are going to be combined with the other classes, and we don't know what order the `super` calls will be made in. The interface is similar to that used for `House` and `Apartment`, which is very useful when we combine the functionality of these four classes in separate subclasses. For example:

```
class HouseRental(Rental, House):
    def prompt_init():
        init = House.prompt_init()
        init.update(Rental.prompt_init())
        return init
    prompt_init = staticmethod(prompt_init)
```

This is slightly surprising, as the class on its own has neither an `__init__` nor `display` method! Because both parent classes appropriately call `super` in these methods, we only have to extend those classes and the classes will behave in the correct order. This is not the case with `prompt_init`, of course, since it is a static method that does not call `super`, so we implement this one explicitly. We should test this class to make sure it is behaving properly before we write the other three combinations:

```
>>> init = HouseRental.prompt_init()
Enter the square feet: 1
Enter number of bedrooms: 2
Enter number of baths: 3
Is the yard fenced?  (yes, no) no
Is there a garage?  (attached, detached, none) none
How many stories? 4
What is the monthly rent? 5
What are the estimated utilities? 6
Is the property furnished?  (yes, no) no
>>> house = HouseRental(**init)
>>> house.display()
PROPERTY DETAILS
================
square footage: 1
```

```
bedrooms: 2
bathrooms: 3

HOUSE DETAILS
# of stories: 4
garage: none
fenced yard: no

RENTAL DETAILS
rent: 5
estimated utilities: 6
furnished: no
```

It looks like it is working fine. The `prompt_init` method is prompting for initializers to all the super classes, and `display()` is also cooperatively calling all three superclasses.

Note

The order of the inherited classes in the preceding example is important. If we had written `class HouseRental(House, Rental)` instead of `class HouseRental(Rental, House)`, `display()` would not have called `Rental.display()`! When `display` is called on our version of `HouseRental`, it refers to the `Rental` version of the method, which calls `super.display()` to get the `House` version, which again calls `super.display()` to get the property version. If we reversed it, `display` would refer to the `House` class's `display()`. When super is called, it calls the method on the `Property` parent class. But `Property` does not have a call to `super` in its `display` method. This means `Rental` class's `display` method would not be called! By placing the inheritance list in the order we did, we ensure that `Rental` calls `super`, which then takes care of the `House` side of the hierarchy. You might think we could have added a `super` call to `Property.display()`, but that will fail because the next superclass of `Property` is `object`, and `object` does not have a `display` method. Another way to fix this is to allow `Rental` and `Purchase` to extend the `Property` class instead of deriving directly from `object`. (Or we could modify the method resolution order dynamically, but that is beyond the scope of this book.)

Now that we have tested it, we are prepared to create the rest of our combined subclasses:

```
class ApartmentRental(Rental, Apartment):
    def prompt_init():
        init = Apartment.prompt_init()
        init.update(Rental.prompt_init())
        return init
    prompt_init = staticmethod(prompt_init)

class ApartmentPurchase(Purchase, Apartment):
    def prompt_init():
        init = Apartment.prompt_init()
        init.update(Purchase.prompt_init())
        return init
    prompt_init = staticmethod(prompt_init)

class HousePurchase(Purchase, House):
    def prompt_init():
        init = House.prompt_init()
        init.update(Purchase.prompt_init())
        return init
    prompt_init = staticmethod(prompt_init)
```

That should be the most intense designing out of our way! Now all we have to do is create the `Agent` class, which is responsible for creating new listings and displaying existing ones. Let's start with the simpler storing and listing of properties:

```
class Agent:
    def __init__(self):
        self.property_list = []

    def display_properties(self):
        for property in self.property_list:
            property.display()
```

Adding a property will require first querying the type of property and whether property is for purchase or rental. We can do this by displaying a simple menu. Once this has been determined, we can extract the correct subclass and prompt for all the details using the `prompt_init` hierarchy we've already developed. Sounds simple? It is. Let's start by adding a dictionary class variable to the `Agent` class:

```
    type_map = {
        ("house", "rental"): HouseRental,
        ("house", "purchase"): HousePurchase,
        ("apartment", "rental"): ApartmentRental,
        ("apartment", "purchase"): ApartmentPurchase
        }
```

That's some pretty funny looking code. This is a dictionary, where the keys are tuples of two distinct strings, and the values are class objects. Class objects? Yes, classes can be passed around, renamed, and stored in containers just like *normal* objects or primitive data types. With this simple dictionary, we can simply hijack our earlier `get_valid_input` method to ensure we get the correct dictionary keys and look up the appropriate class, like this:

```
    def add_property(self):
        property_type = get_valid_input(
                "What type of property? ",
                ("house", "apartment")).lower()
        payment_type = get_valid_input(
                "What payment type? ",
                ("purchase", "rental")).lower()

        PropertyClass = self.type_map[
            (property_type, payment_type)]
        init_args = PropertyClass.prompt_init()
        self.property_list.append(PropertyClass(**init_args))
```

This may look a bit funny too! We look up the class in the dictionary and store it in a variable named `PropertyClass`. We don't know exactly which class is available, but the class knows itself, so we can polymorphically call `prompt_init` to get a dictionary of values appropriate to pass into the constructor. Then we use the keyword argument syntax to convert the dictionary into arguments and construct the new object to load the correct data.

Now our user can use this `Agent` class to add and view lists of properties. It wouldn't take much work to add features to mark a property as available or unavailable or to edit and remove properties. Our prototype is now in a good enough state to take to a real estate `agent` and demonstrate its functionality. Here's how a demo session might work:

```
>>> agent = Agent()
>>> agent.add_property()
What type of property?  (house, apartment) house
What payment type?  (purchase, rental) rental
Enter the square feet: 900
Enter number of bedrooms: 2
Enter number of baths: one and a half
Is the yard fenced?  (yes, no) yes
Is there a garage?  (attached, detached, none) detached
How many stories? 1
What is the monthly rent? 1200
What are the estimated utilities? included
```

```
Is the property furnished?  (yes, no) no
>>> agent.add_property()
What type of property?  (house, apartment) apartment
What payment type?  (purchase, rental) purchase
Enter the square feet: 800
Enter number of bedrooms: 3
Enter number of baths: 2
What laundry facilities does the property have?  (coin, ensuite,
one) ensuite
Does the property have a balcony? (yes, no, solarium) yes
What is the selling price? $200,000
What are the estimated taxes? 1500
>>> agent.display_properties()
PROPERTY DETAILS
================
square footage: 900
bedrooms: 2
bathrooms: one and a half

HOUSE DETAILS
# of stories: 1
garage: detached
fenced yard: yes
RENTAL DETAILS
rent: 1200
estimated utilities: included
furnished: no
PROPERTY DETAILS
================
square footage: 800
bedrooms: 3
bathrooms: 2

APARTMENT DETAILS
laundry: ensuite
has balcony: yes
PURCHASE DETAILS
selling price: $200,000
estimated taxes: 1500
```