Lecture 6: Classes

Adding a little class to your code.

Python Classes

A “class” is a blueprint for creating objects.

As an object-oriented programming language, virtually everything in Python is an object.

Python Classes

A “class” is a blueprint for creating objects.

As an object-oriented programming language, virtually everything in Python is an object.

Classes can have properties (variables)
Classes can have methods (functions)

Python Classes

A “class” is a blueprint for creating objects.

As an object-oriented programming language, virtually everything in Python is an object.

Classes can have properties (variables)
Classes can have methods (functions)

Creating an object using the class “blueprint” is called an “instance” of the class

Classes are Blueprints

Class

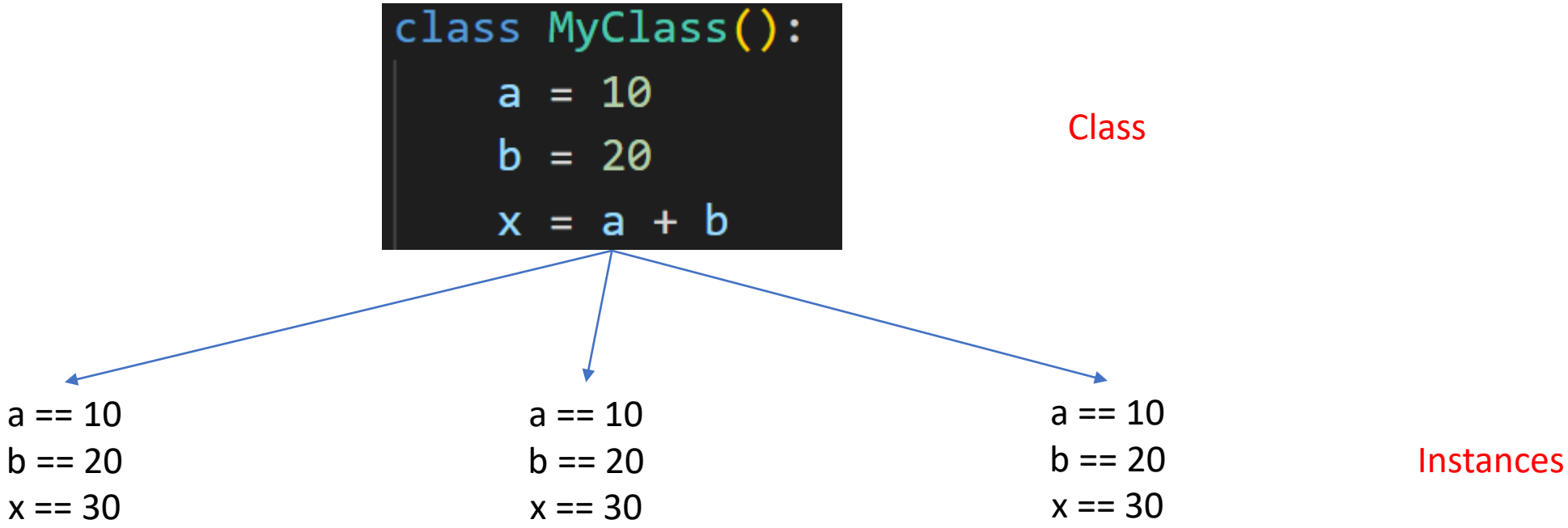
Brown siding
Hard wood floors
White paint

Yellow siding
Vinyl floors
White paint

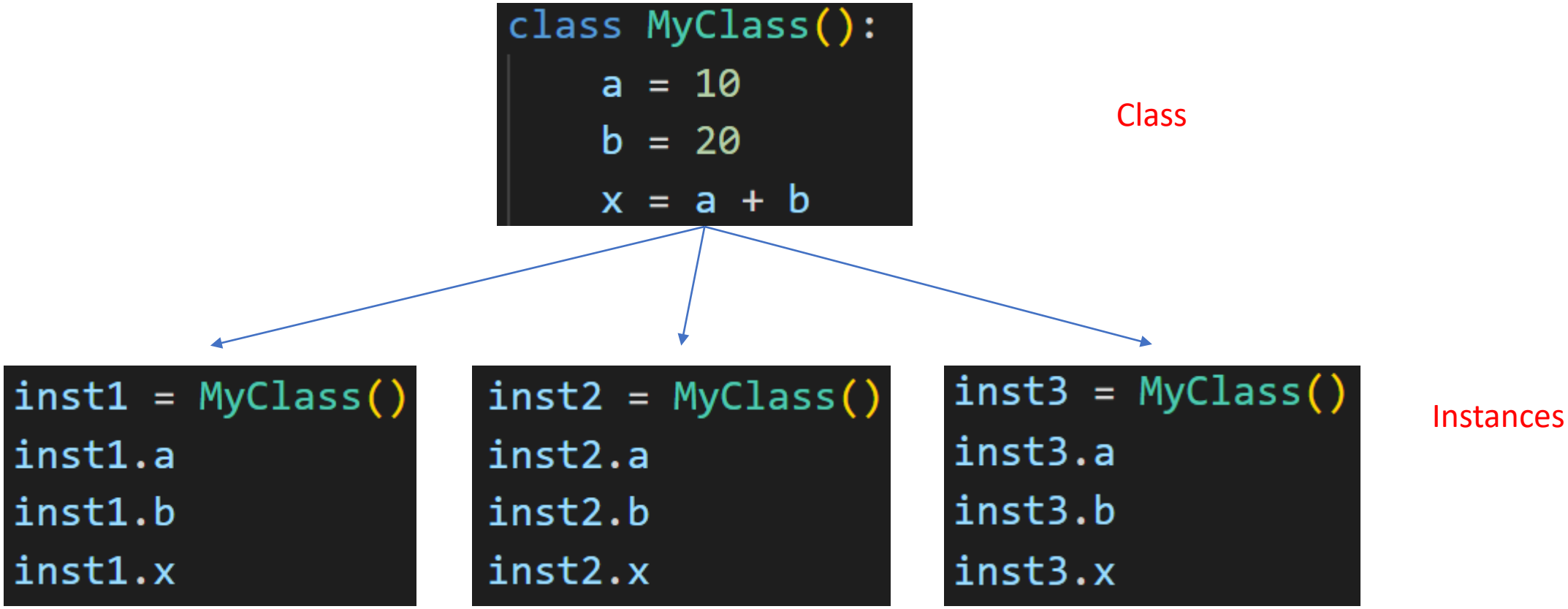
Gray siding
Hard wood floors
Blue paint

Instances

A Simple, Static Class



A Simple, Static Class



Customizing Instances

```
class MyClass():
    def __init__(self, a, b):
        self.a = a
        self.b = b
        self.x = a + b
```

Customizing Instances

```
class MyClass():
    def __init__(self, a, b):
        self.a = a
        self.b = b
        self.x = a + b
```

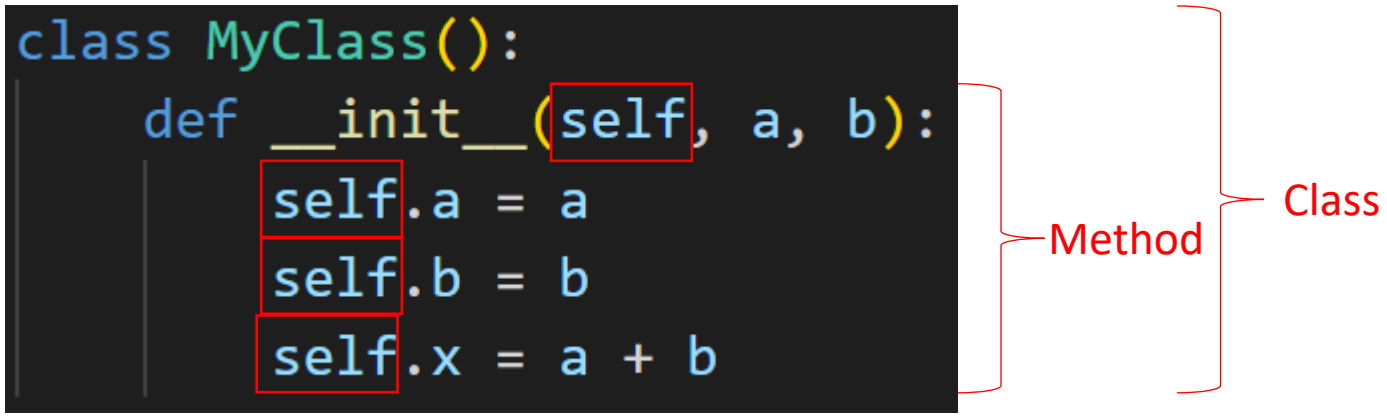
Method

Class

```
def __init__(self, a, b):
    x = a + b
    return x
```

Function

What is “self”?



- A standard positional argument
- Can be named anything, but is named “self” by convention
- Points to an instance made from this class

Methods always* reserve the first positional argument for this!

Customizing Instances

```python
class MyClass():
    def __init__(self, a, b):
        self.a = a
        self.b = b
        self.x = a + b
```

Class

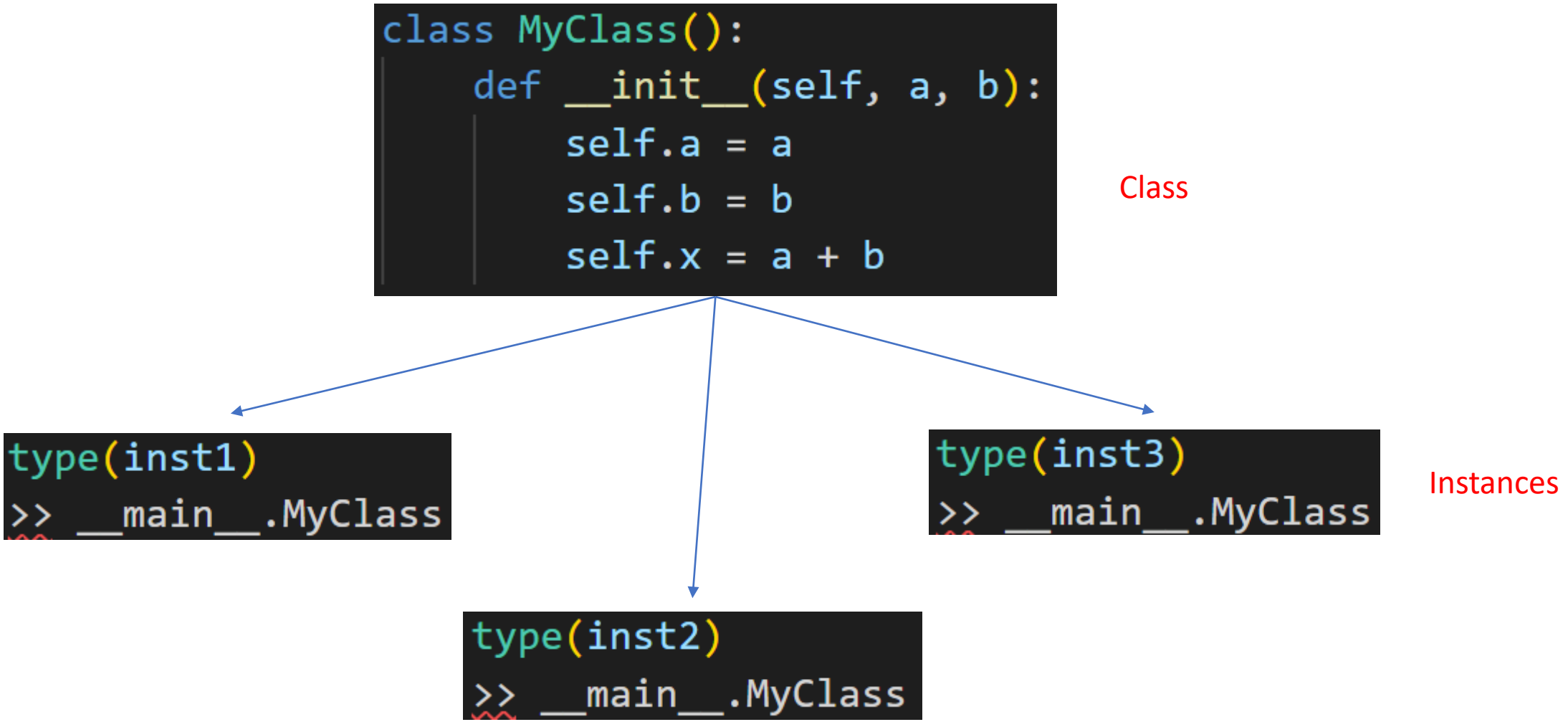
```python
inst1 = MyClass(10, 20)
inst1.x
>> 30
```

```python
inst2 = MyClass(1, 2)
inst2.x
>> 3
```

```python
inst3 = MyClass(5, 5)
inst3.x
>> 10
```

Instances

Instances created from the same class



All Objects in Python are Classes/Instances

```
x = {'a':0}
type(x)
>> dict
```

```
x = 'Hello world!'
type(x)
>> str
```

```
x = [1, 2]
type(x)
>> list
```

```
type(inst1)
>> __main__.MyClass
```

```
type(inst2)
>> __main__.MyClass
```

```
type(inst3)
>> __main__.MyClass
```

Example: Adding methods to a class

Create a class that starts with values for the number of bedrooms, bathrooms, and the square footage.

Add a method that uses these three values to calculate a home price.

Then add a method that randomly picks a markup for the neighborhood and modifies the result

First, plan your class out!

```
class HouseValues():
    def __init__(self):
        # needs three values: num bedrooms, num bathrooms, sqft
        pass
```

First, plan your class out!

```
class HouseValues():
    def __init__(self):
        # needs three values: num bedrooms, num bathrooms, sqft
        pass

    def estimate_value(self):
        # use an equation of questionable accuracy that I found on a random
        # website to estimate the value based on those three parameters
        pass
```

First, plan your class out!

```
class HouseValues():
    def __init__(self):
        # needs three values: num bedrooms, num bathrooms, sqft
        pass

    def estimate_value(self):
        # use an equation of questionable accuracy that I found on a random
        # website to estimate the value based on those three parameters
        pass

    def pick_a_neighborhood(self):
        # randomly pick a modifier to multiply the value estimate by
        pass
```

Filling out the init method

```
class HouseValues():
    def __init__(self, num_bedrooms, num_baths, sqft):
        self.num_bedrooms = num_bedrooms
        self.num_baths = num_baths
        self.sqft = sqft
```

Filling out the init method

```
class HouseValues():
    def __init__(self, num_bedrooms, num_baths, sqft):
        self.num_bedrooms = num_bedrooms
        self.num_baths = num_baths
        self.sqft = sqft
```

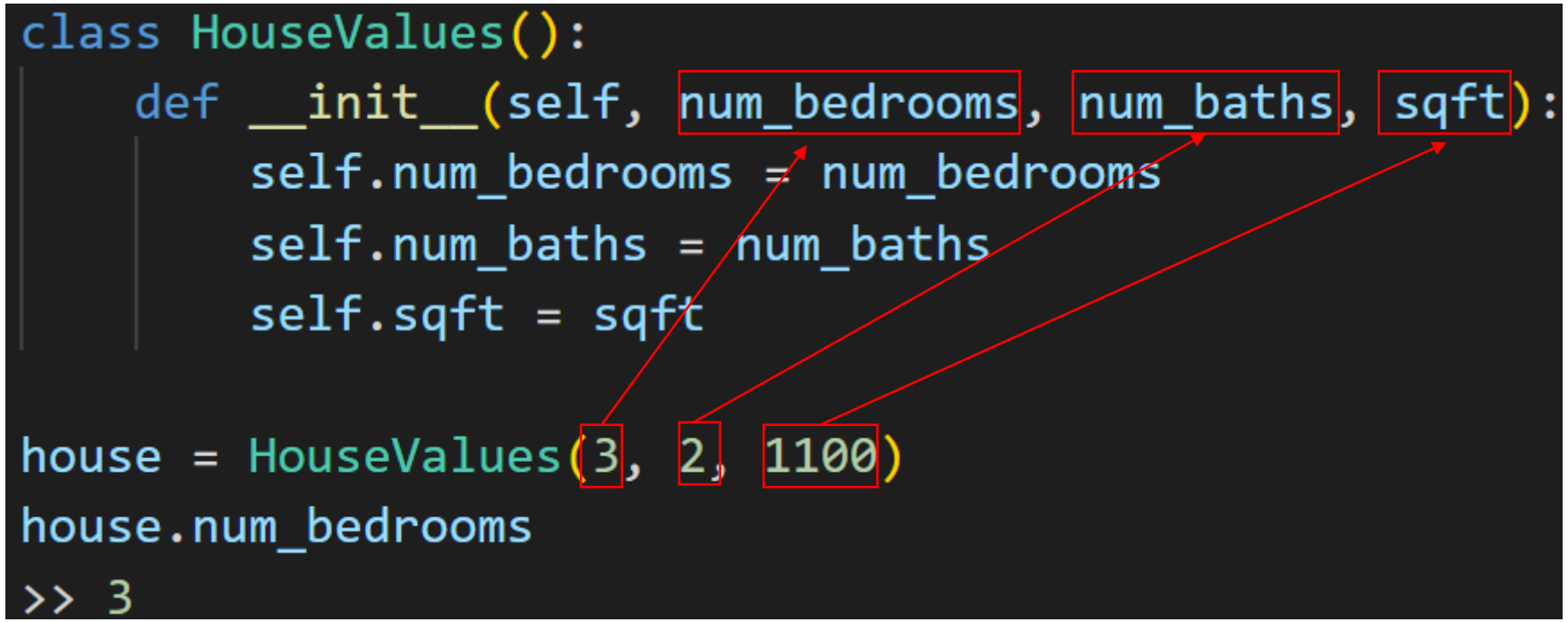
This requires 3 positional arguments when called. Why not 4?

Filling out the init method

```
class HouseValues():
    def __init__(self, num_bedrooms, num_baths, sqft):
        self.num_bedrooms = num_bedrooms
        self.num_baths = num_baths
        self.sqft = sqft

house = HouseValues(3, 2, 1100)
house.num_bedrooms
>> 3
```

Follow the variables



What happens if we don't use self?

```
class HouseValues():
    def __init__(self, num_bedrooms, num_baths, sqft):
        num_bedrooms = num_bedrooms
        num_baths = num_baths
        sqft = sqft

house = HouseValues(3, 2, 1100)
house.num_bedrooms
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
Cell In [235], line 2
      1 house = HouseValues(3, 2, 1100)
----> 2

AttributeError: 'HouseValues' object has no attribute 'num_bedrooms'
```

Build out the next method

```
class HouseValues():
    def __init__(self, num_bedrooms, num_baths, sqft):
        self.num_bedrooms = num_bedrooms
        self.num_baths = num_baths
        self.sqft = sqft

    def estimate_value(self):
        #add 10% per num of bedrooms over 1
        bedroom_mod = ((self.num_bedrooms - 1) * 0.1) + 1

        #add 5% per num of baths over 1
        bath_mod = ((self.num_baths - 1) * 0.05) + 1

        self.value = (self.sqft * 400) * bedroom_mod * bath_mod
        print(f'I estimate this house will be worth ${round(self.value, 2)}')
```

```
house1 = HouseValues(2, 2, 950)
house1.estimate_value()
```

```
I estimate this house will be worth $438900.0
```

Follow the Variables

```
class HouseValues():
    def __init__(self, num_bedrooms, num_baths, sqft):
        self.num_bedrooms = num_bedrooms
        self.num_baths = num_baths
        self.sqft = sqft

    def estimate_value(self):
        #add 10% per num of bedrooms over 1
        bedroom_mod = ((self.num_bedrooms - 1) * 0.1) + 1

        #add 5% per num of baths over 1
        bath_mod = ((self.num_baths - 1) * 0.05) + 1

        self.value = (self.sqft * 400) * bedroom_mod * bath_mod
        print(f'I estimate this house will be worth ${round(self.value, 2)}')
```

```
house1 = HouseValues(2, 2, 950)
house1.estimate_value()
```

1. Pass 950 into the third positional argument

```
I estimate this house will be worth $438900.0
```

Follow the Variables

```
class HouseValues():
    def __init__(self, num_bedrooms, num_baths, sqft):
        self.num_bedrooms = num_bedrooms
        self.num_baths = num_baths
        self.sqft = sqft

    def estimate_value(self):
        #add 10% per num of bedrooms over 1
        bedroom_mod = ((self.num_bedrooms - 1) * 0.1) + 1

        #add 5% per num of baths over 1
        bath_mod = ((self.num_baths - 1) * 0.05) + 1

        self.value = (self.sqft * 400) * bedroom_mod * bath_mod
        print(f'I estimate this house will be worth ${round(self.value, 2)}')
```

2. Store sqft in the sqft attribute of any instances created from this class

```
house1 = HouseValues(2, 2, 950)
house1.estimate_value()
```

1. Pass 950 into the third positional argument

```
I estimate this house will be worth $438900.0
```

Follow the Variables

```
class HouseValues():
    def __init__(self, num_bedrooms, num_baths, sqft):
        self.num_bedrooms = num_bedrooms
        self.num_baths = num_baths
        self.sqft = sqft

    def estimate_value(self):
        #add 10% per num of bedrooms over 1
        bedroom_mod = ((self.num_bedrooms - 1) * 0.1) + 1

        #add 5% per num of baths over 1
        bath_mod = ((self.num_baths - 1) * 0.05) + 1

        self.value = (self.sqft * 400) * bedroom_mod * bath_mod
        print(f'I estimate this house will be worth ${round(self.value, 2)}')
```

2. Store sqft in the sqft attribute of any instances created from this class

3. Reference the attribute in other methods of this class

```
house1 = HouseValues(2, 2, 950)
house1.estimate_value()
```

1. Pass 950 into the third positional argument

```
I estimate this house will be worth $438900.0
```

Finish the Last Method

```
def pick_a_neighborhood(self):
    value = random.normal(1, 0.1)
    if value > 1.2:
        print('Whoa, you got an expensive neighborhood!')
    elif value > 1:
        print('Fairly pricy neighborhood.')
    elif value < 1:
        print('Maybe not the nicest neighborhood.')
    return value
```

```
n_mod = self.pick_a_neighborhood()

self.value = (self.sqft * 400) * bedroom_mod * bath_mod * n_mod
```

Modify the estimate_value method to use it

Final Test

```
house1 = HouseValues(2, 2, 950)
house2 = HouseValues(1, 1, 700)

house1.estimate_value()
>> Fairly pricy neighborhood.
>> I estimate this house will be worth $516607.27

house2.estimate_value()
>> Maybe not the nicest neighborhood.
>> I estimate this house will be worth $273543.39
```

Overview

Functions

1. Used for repetition, to avoid copy-paste
2. Used to organize code into logical groupings

Classes

1. Creates a blueprint we can use over and over again
2. Stores attributes (like variables)
3. Stores methods (like functions)