Object-Oriented Programming

April 12, 2022

Learning Goals

After today, students will be able to:

- Compare and contrast the efficacy of using different data structures for a program written in Python.
- Design and implement custom Python objects (classes) for Python programs to augment Python’s object functionalities.

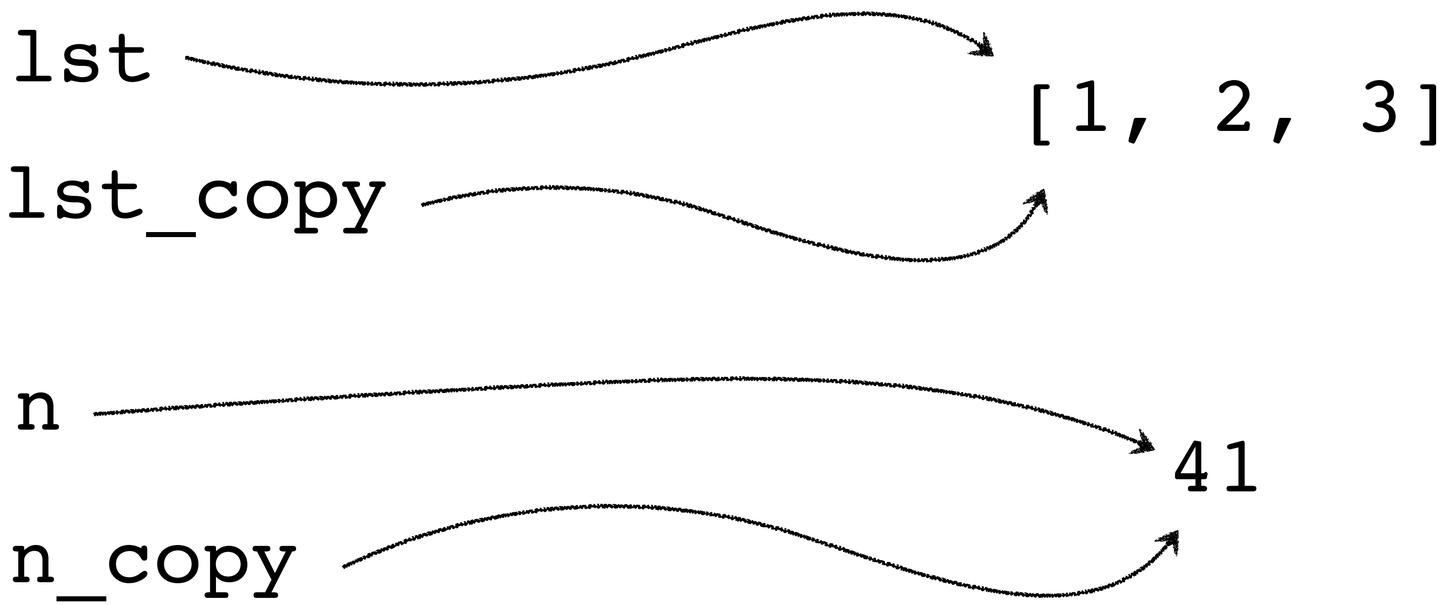
Data Model Diagram Activity

On the whiteboard, pick a section or concept from the data model handout and **create a visual representation to help explain the concept**. As a reminder, the more substantial sections were:

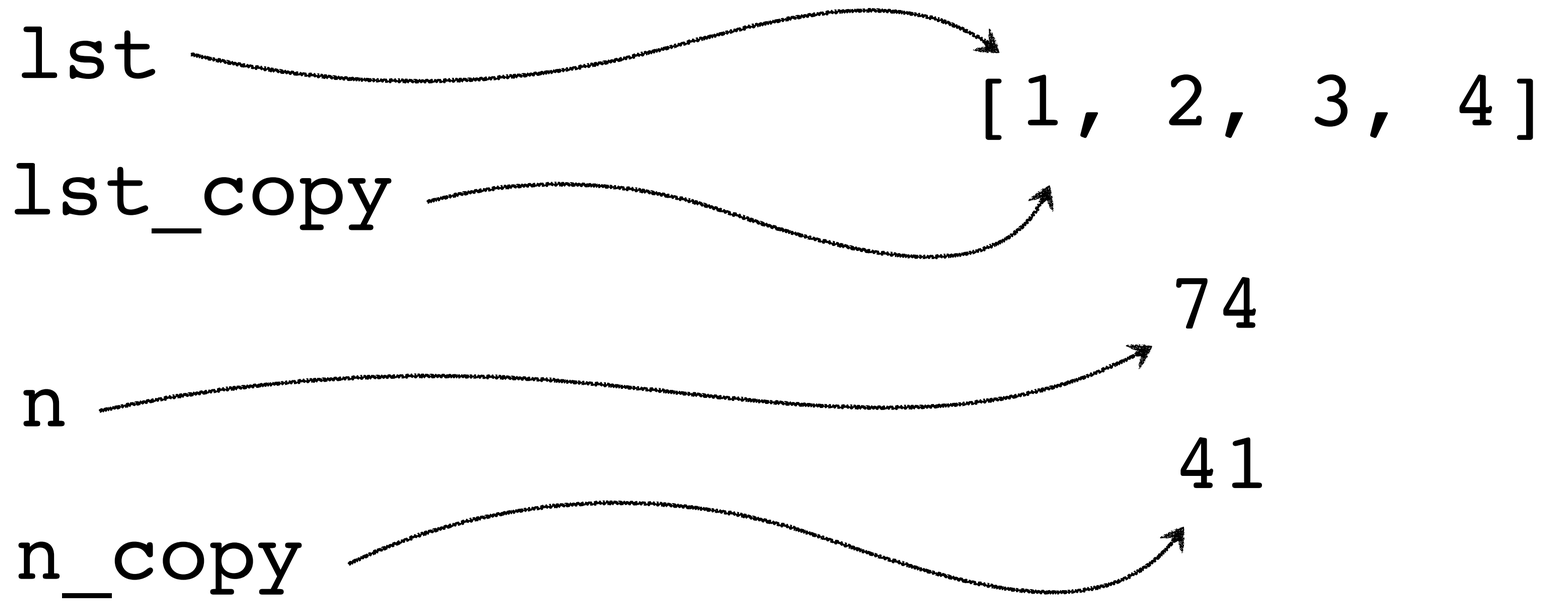
- Objects have value, type, and identity
- Objects and mutability
- Modifying an object from a function
- Nested objects and copies

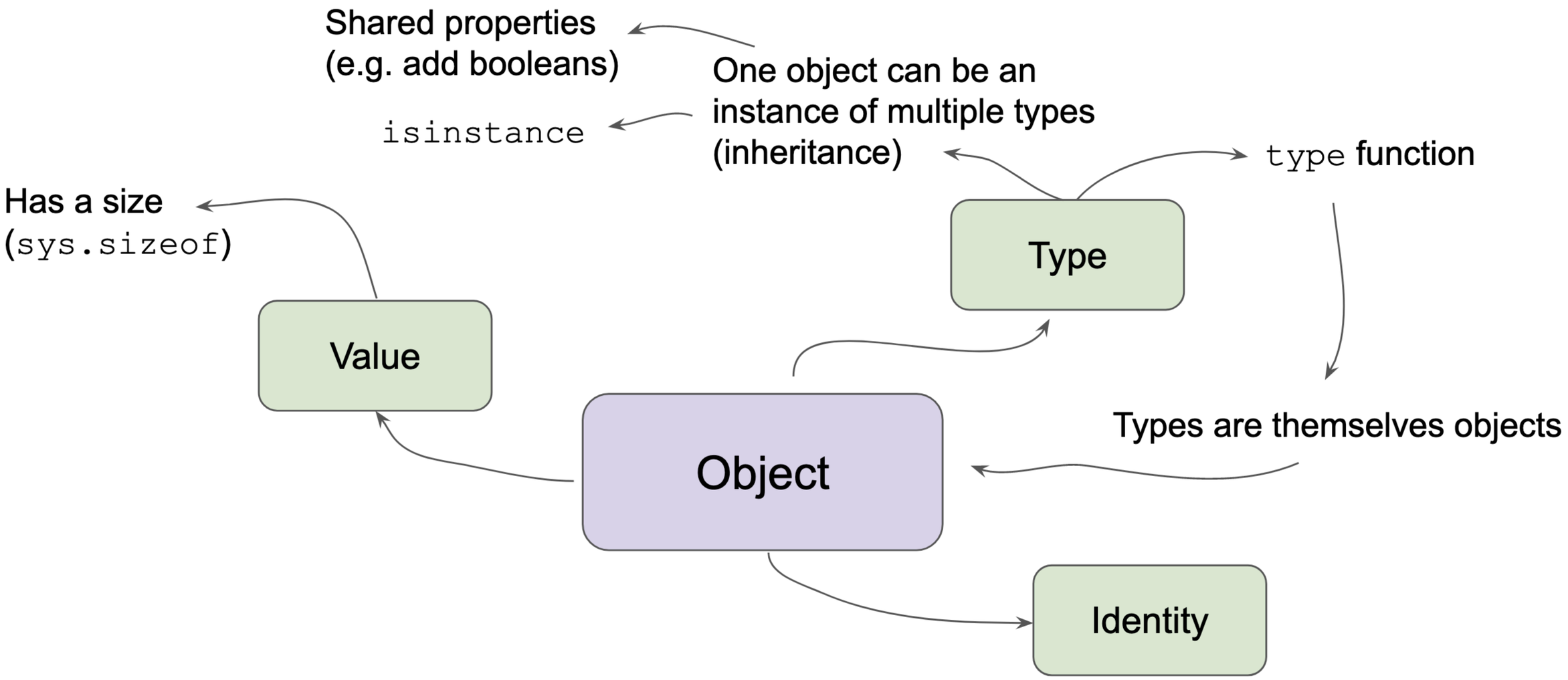
You’re welcome to make a diagram that fully explains the concept you’re tackling, but you can also think of the diagram as a supplement to the handout.

```
lst = [1, 2, 3]
lst_copy = lst

n = 41
n_copy = n
```




```
lst += [4]

n += 33
```




Shared properties
(e.g. add booleans)
One object can be an
instance of multiple types
(inheritance)
isinstance
type function
Has a size
(sys.sizeof)
Type
Value
Object
Types are themselves objects
Identity

Let's build a program to help Stanford manage students and courses...

In addition, we'd like a function to enroll a student in a course, which should also check if the student has the necessary prerequisites for the class.

What data structures would you use to solve this problem?

(What would you use to represent a course? A student? How would you implement the enroll function?)

Let's build a program to help Stanford manage students and courses...

Every ***student*** has...

- Name (string)
- SUNet ID (string)
- Collection of courses they've taken in the past
  - Grades they received in those courses
- Collection of courses they're currently taking

In addition, we'd like a function to enroll a student in a course, which should also check if the student has the necessary prerequisites for the class.

What data structures would you use to solve this problem?

(What would you use to represent a course? A student? How would you implement the enroll function?)

Let's build a program to help Stanford manage students and courses...

Every ***student*** has...

- Name (string)
- SUNet ID (string)
- Collection of courses they've taken in the past
  - Grades they received in those courses
- Collection of courses they're currently taking

Every ***course*** has...

- Department (string)
- Course number (string)
- Quarter (string)
- Collection of prerequisites
- Collection of students who are currently enrolled

In addition, we'd like a function to enroll a student in a course, which should also check if the student has the necessary prerequisites for the class.

What data structures would you use to solve this problem?

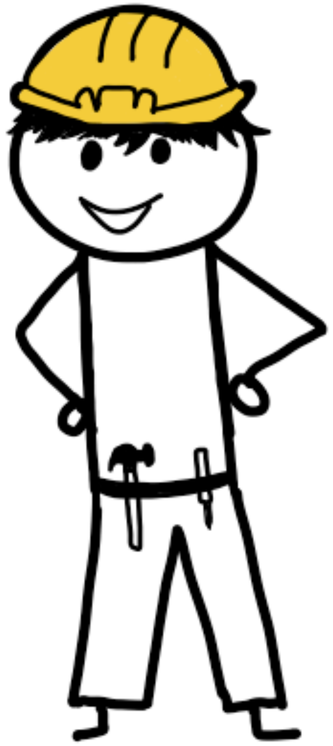
(What would you use to represent a course? A student? How would you implement the enroll function?)

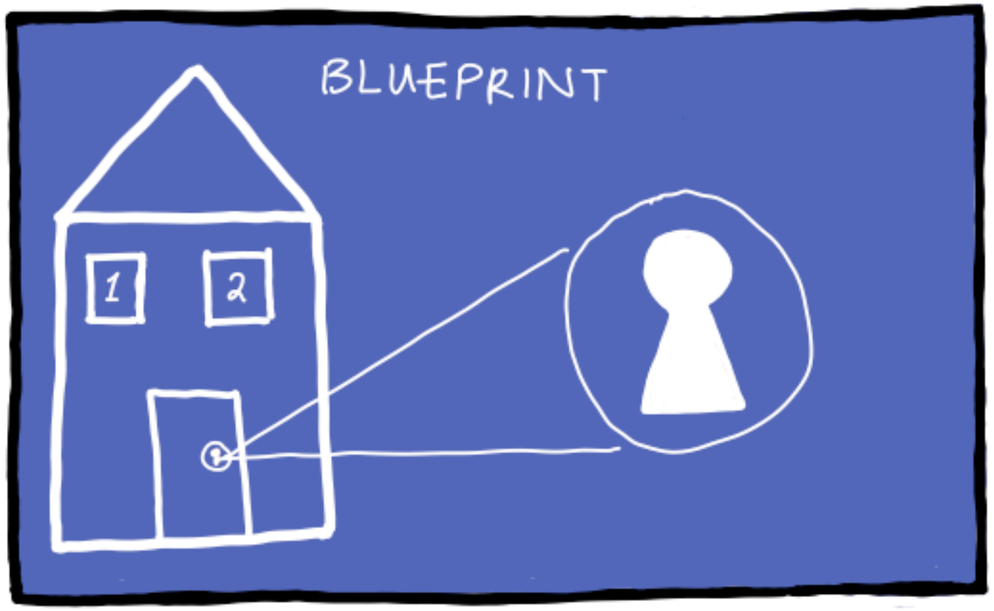
Classes

High-Level

Imagine I'm opening a residential construction company which is going to build several houses...

First, I need a blueprint for a house. This is the **class object**.

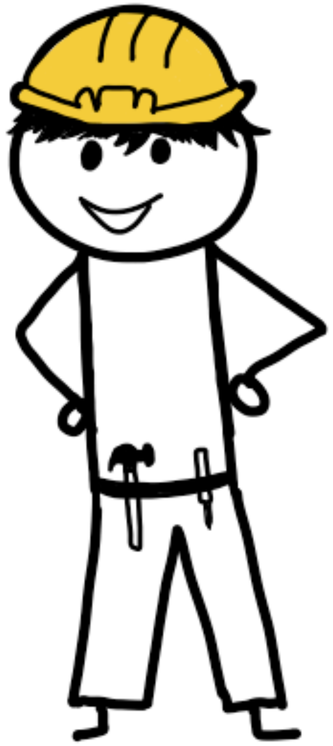


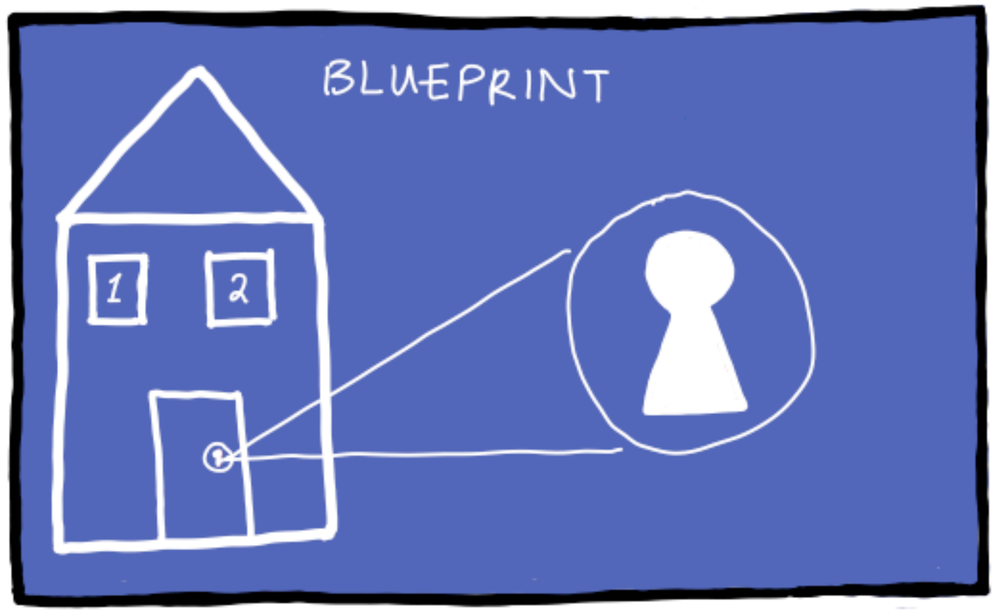


High-Level

Imagine I'm opening a residential construction company which is going to build several houses...

First, I need a blueprint for a house. This is the **class object**.



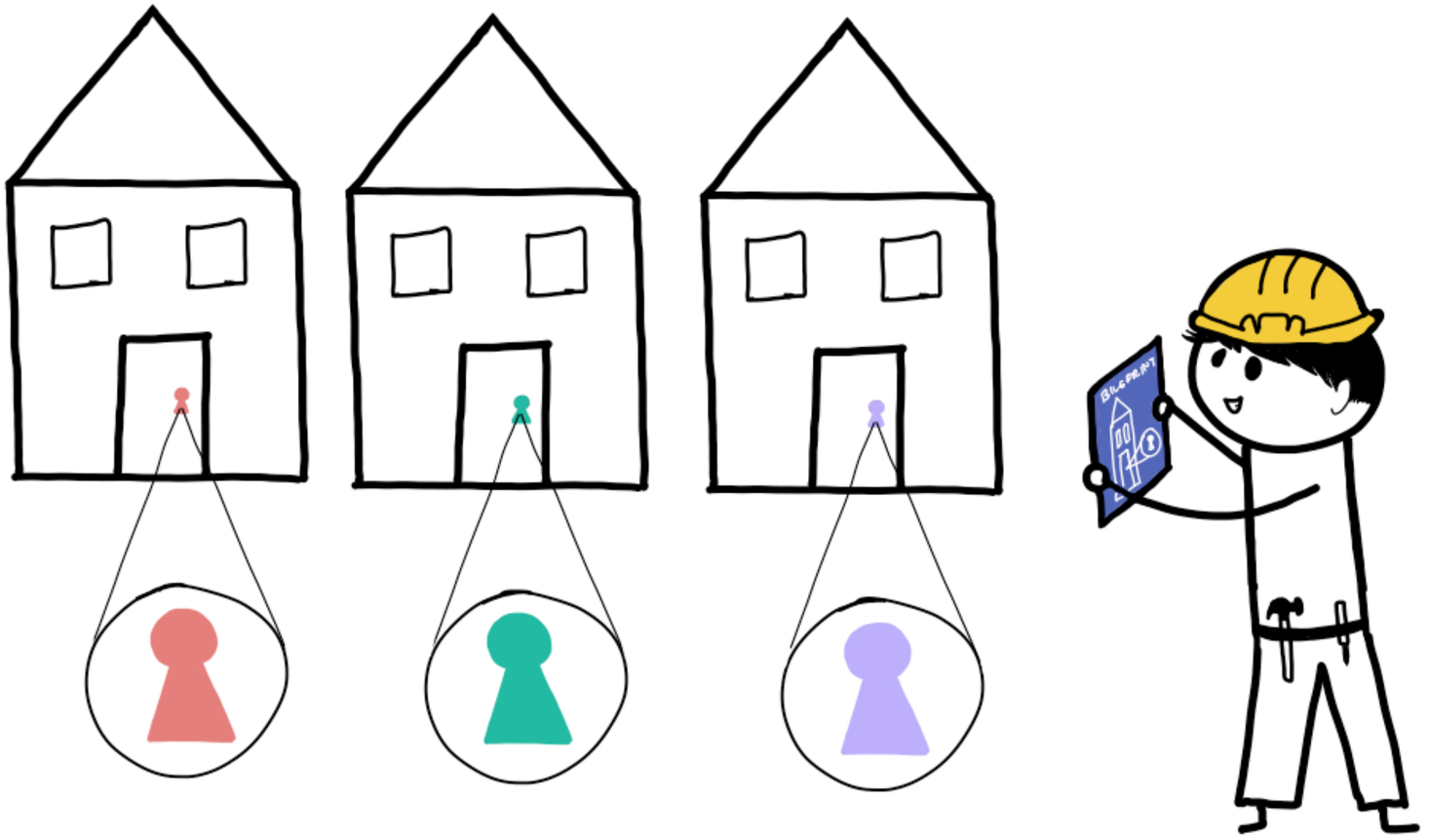


✨ btw y'all, my sister made these! 💜

High-Level

Then, I can use that blueprint to build several houses. Some properties of the houses will be the same and others will be different.

Each house is **an instance (object)** of the class.



High-Level

The blueprint for a house

```
class House:
    utilities = {
        'electricity': 'A&E #8675309',
        'water': 'Palo Alto Mutual #6054756961'
    }

    def __init__(self):
        self.locked = True
```

High-Level

The blueprint for a house

```
class House:
    utilities = {
        'electricity': 'A&E #8675309',
        'water': 'Palo Alto Mutual #6054756961'
    }

    def __init__(self):
        self.locked = True
```

These attributes are shared among the instances (houses)

High-Level

The blueprint for a house

```
class House:
    utilities = {
        'electricity': 'A&E #8675309',
        'water': 'Palo Alto Mutual #6054756961'
    }

    def __init__(self):
        self.locked = True
```

These attributes are shared among the instances (houses)

This is run every time an instance is declared and sets up instance-specific properties (it's the "constructor")

High-Level

```
class House:
    utilities = {
        'electricity': 'A&E #8675309',
        'water': 'Palo Alto Mutual #6054756961'
    }

    def __init__(self):
        self.locked = True
```

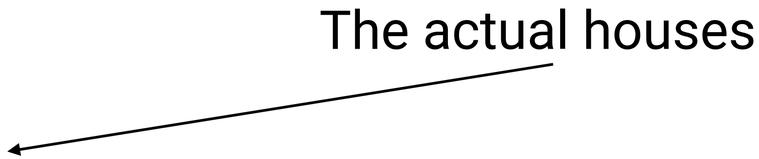
High-Level

The actual houses


```
red = House()
blue = House()
green = House()
```

```
class House:
    utilities = {
        'electricity': 'A&E #8675309',
        'water': 'Palo Alto Mutual #6054756961'
    }

    def __init__(self):
        self.locked = True
```

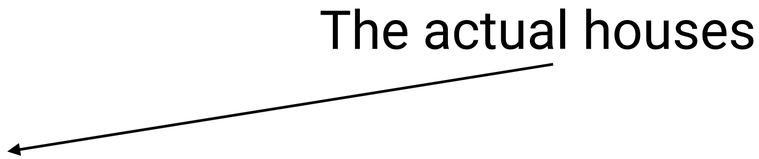
High-Level




```
red = House()
blue = House()
green = House()

House.utilities['electricity'] # => 'A&E #8675309'
red.utilities['electricity']   # => 'A&E #8675309'
green.utilities['electricity'] # => 'A&E #8675309'
```

```
class House:
    utilities = {
        'electricity': 'A&E #8675309',
        'water': 'Palo Alto Mutual #6054756961'
    }

    def __init__(self):
        self.locked = True
```

High-Level




```
red = House()
blue = House()
green = House()

House.utilities['electricity'] # => 'A&E #8675309'
red.utilities['electricity']   # => 'A&E #8675309'
green.utilities['electricity'] # => 'A&E #8675309'

red.locked  # => True
blue.locked # => True
```

```
class House:
    utilities = {
        'electricity': 'A&E #8675309',
        'water': 'Palo Alto Mutual #6054756961'
    }

    def __init__(self):
        self.locked = True
```

High-Level

The actual houses

```
red = House()
blue = House()
green = House()

House.utilities['electricity'] # => 'A&E #8675309'
red.utilities['electricity']   # => 'A&E #8675309'
green.utilities['electricity'] # => 'A&E #8675309'

red.locked  # => True
blue.locked # => True

red.locked = False
blue.locked # => True
```

Note: In Python, all attributes are public

```
class House:
    utilities = {
        'electricity': 'A&E #8675309',
        'water': 'Palo Alto Mutual #6054756961'
    }

    def __init__(self):
        self.locked = True
```

But wait... what's `self`?

```
class House:
    def __init__(self):
        self.locked = True
```

But wait... what's `self`?

```
class House:
    def __init__(self):
        self.locked = True
```

When the function is run on a class instance, the first parameter to every method is a reference to the object itself. It could be named anything, but `self` is the traditional name.

But wait... what's `self`?

```
class House:
    def __init__(self):
        self.locked = True
```

When the function is run on a class instance, the first parameter to every method is a reference to the object itself. It could be named anything, but `self` is the traditional name.

```
House.__init__  # => <function __init__(self)>

red = House()
red.__init__  # => <bound method House.__init__>
```

But wait... what's `self`?

```
class House:
    def __init__(self):
        self.locked = True
```

When the function is run on a class instance, the first parameter to every method is a reference to the object itself. It could be named anything, but `self` is the traditional name.

```
House.__init__  # => <function __init__(self)>

red = House()
red.__init__  # => <bound method House.__init__>
```

This applies to other methods as well, not just `__init__`.

```
instance.method(some args) ~ function(instance, some args)
```

Custom Instantiation

```
class Student:
    def __init__(self, name, sunet):
        self.name = name.title()

        # validate the SUNet
        if not set(sunet) <= set('0123456789'):
            raise ValueError(f"Invalid SUNet: {sunet}.")
        self.sunet = sunet
```

Just like a normal function, `__init__` can have parameters!

Custom Instantiation

```
class Student:
    def __init__(self, name, sunet):
        self.name = name.title()

        # validate the SUNet
        if not set(sunet) <= set('0123456789'):
            raise ValueError(f"Invalid SUNet: {sunet}.")
        self.sunet = sunet
```

Just like a normal function, `__init__` can have parameters!

```
parth = Student('parth sarin', 'noneya') # ValueError
```

Custom Instantiation

```
class Student:
    def __init__(self, name, sunet):
        self.name = name.title()

        # validate the SUNet
        if not set(sunet) <= set('0123456789'):
            raise ValueError(f"Invalid SUNet: {sunet}.")
        self.sunet = sunet
```

Just like a normal function, `__init__` can have parameters!

```
parth = Student('parth sarin', 'noneya') # ValueError

tara = Student('tara jones', '5625165')
tara.name # => 'Tara Jones'
```

Magic Methods

Python Uses Magic Methods!

```
str(x)   # => x.__str__()

x == y   # => x.__eq__(y)
x < y    # => x.__lt__(y)
x + y    # => x.__add__(y)

next(x)  # => x.__next__()
len(x)   # => x.__len__()
hash(x)  # => x.__hash__()

el in x # => x.__contains__(el)
```

Full list here!