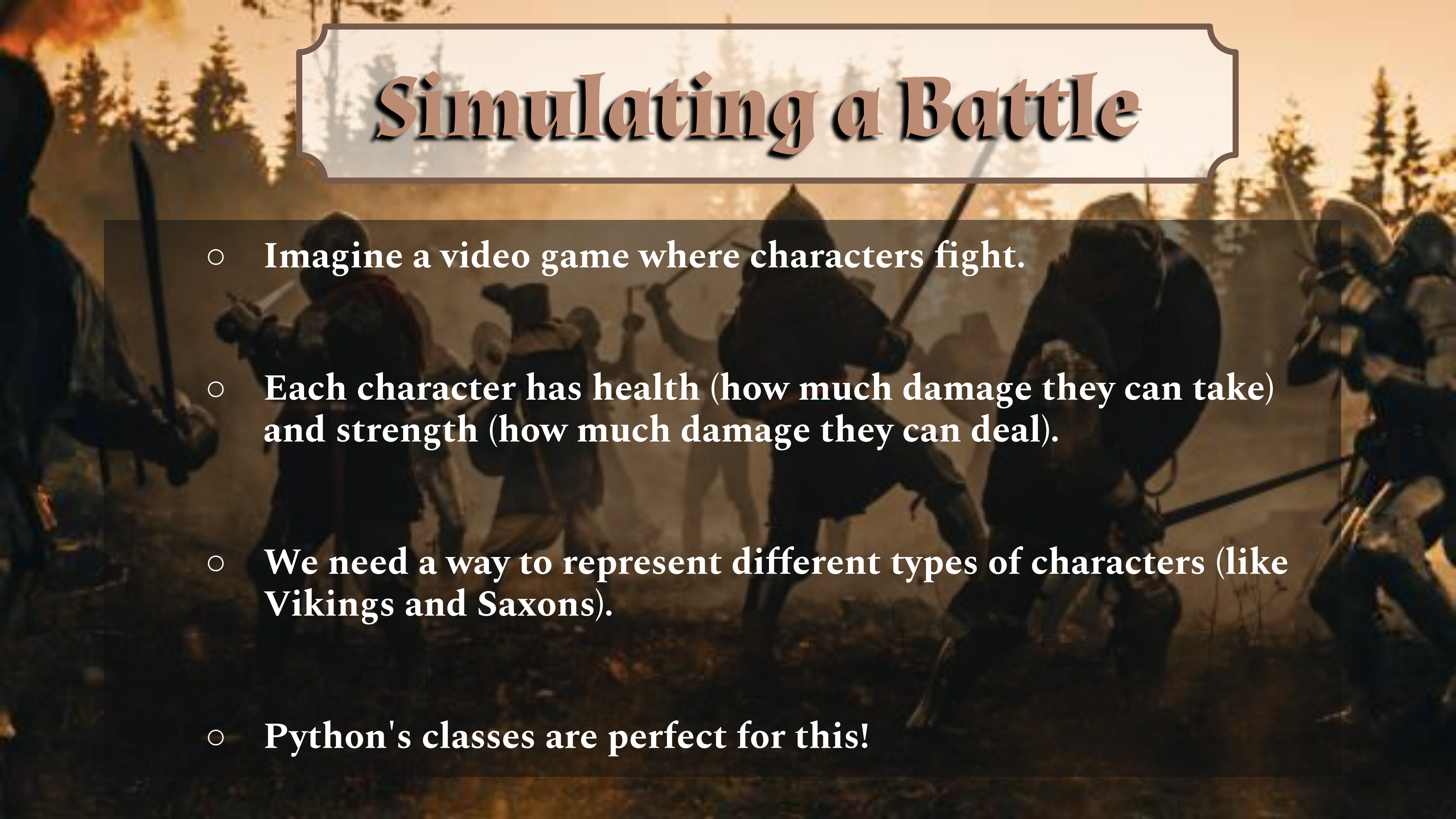
A Clash of Classes: Simulating War in Python!

Vikings-Mini-Projec t

Jithin Kumar
Katherine Torian
Anirudh Unni



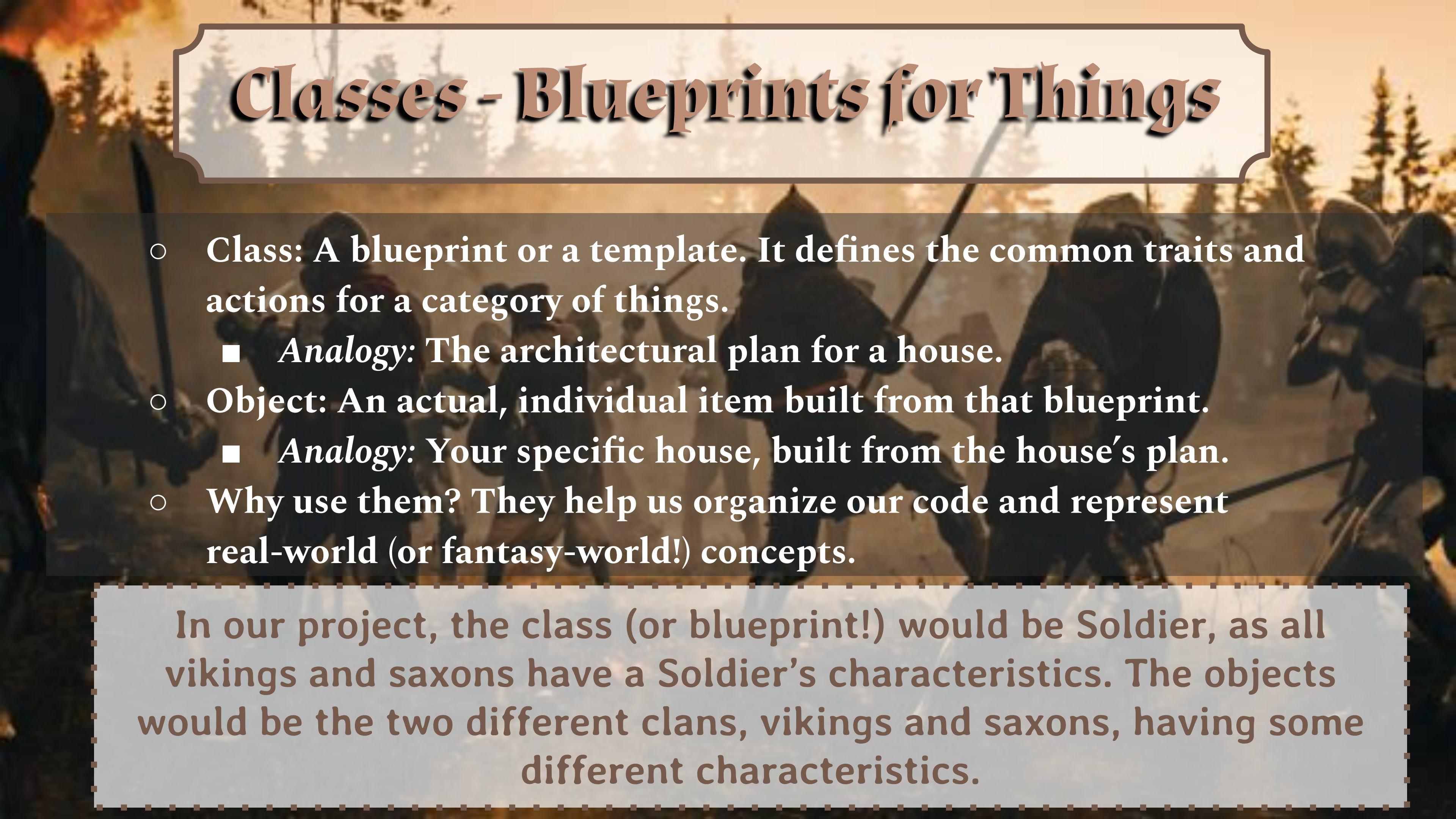
Simulating a Battle

- Imagine a video game where characters fight.
- Each character has health (how much damage they can take) and strength (how much damage they can deal).
- We need a way to represent different types of characters (like Vikings and Saxons).
- Python's classes are perfect for this!

Classes - Blueprints for Things

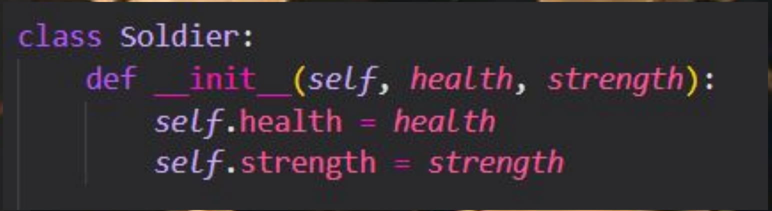
- Class: A blueprint or a template. It defines the common traits and actions for a category of things.
  - *Analogy:* The architectural plan for a house.
- Object: An actual, individual item built from that blueprint.
  - *Analogy:* Your specific house, built from the house's plan.
- Why use them? They help us organize our code and represent real-world (or fantasy-world!) concepts.

In our project, the class (or blueprint!) would be Soldier, as all vikings and saxons have a Soldier's characteristics. The objects would be the two different clans, vikings and saxons, having some different characteristics.

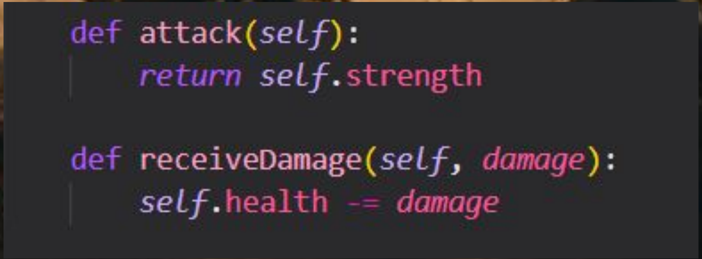


Our Basic Fighter: The Soldier

- This is the most fundamental blueprint for *any* fighter in our game. This means this is our parent class.
- <u>What it HAS (Attributes/Data):</u>
  - health: How much life they have.
  - strength: How powerful their attacks are.
- <u>What it CAN DO (Methods/Actions):</u>
  - attack(): Returns how much damage they deal, directly related to the 'strength' attribute.
  - receiveDamage(damage): Lowers their health when being hit, related to the 'health' attribute.



```python
class Soldier:
    def __init__(self, health, strength):
        self.health = health
        self.strength = strength
```



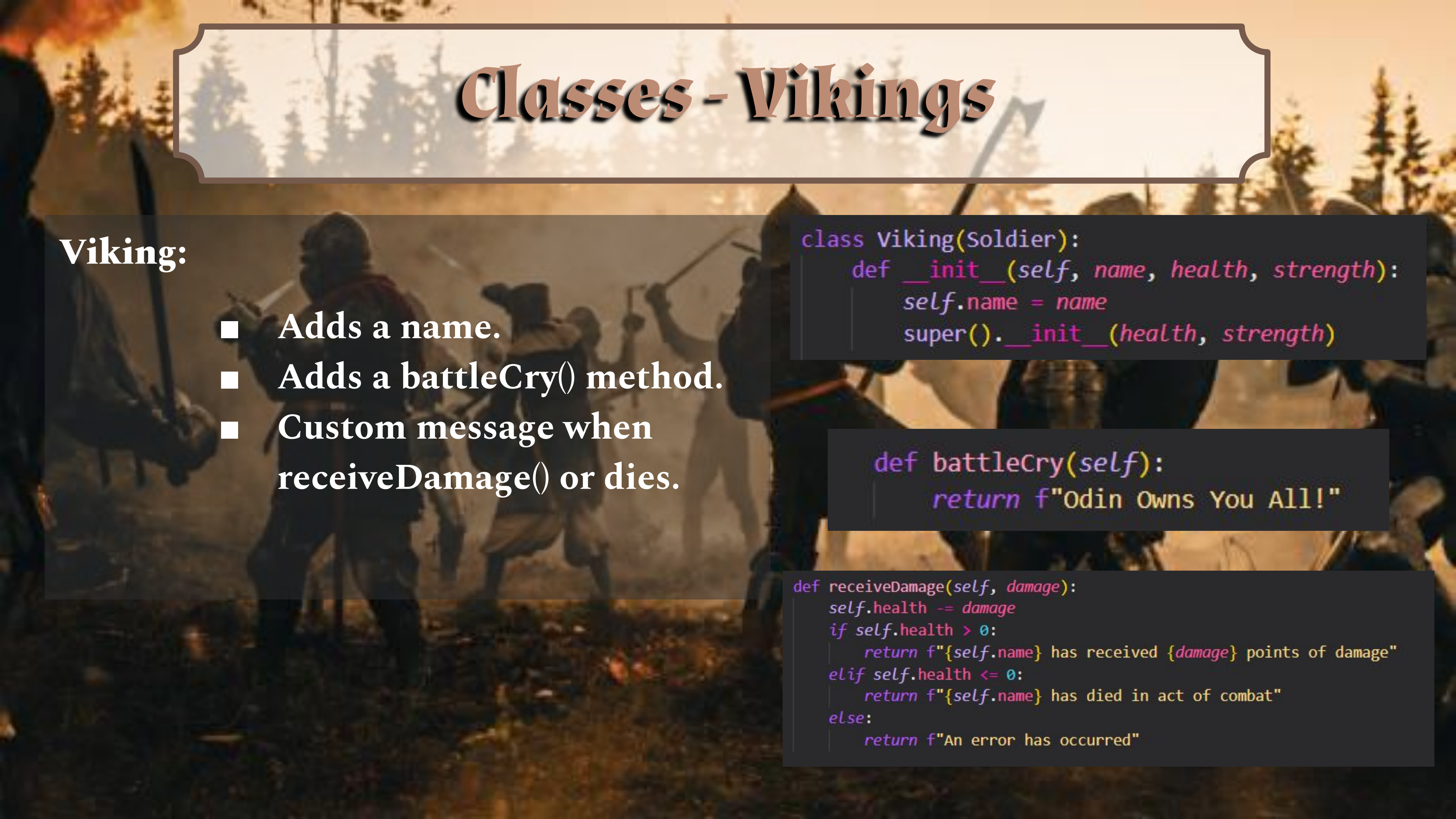
```python
    def attack(self):
        return self.strength

    def receiveDamage(self, damage):
        self.health -= damage
```

Specialized Fighters: Viking and Saxon (Inheritance!)

Main Points:

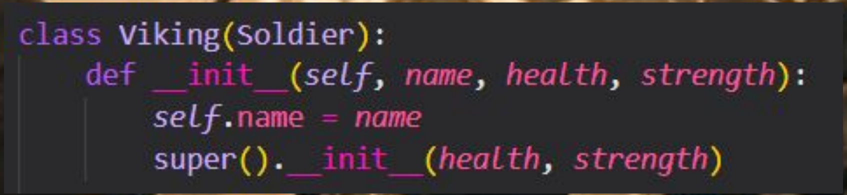
- Instead of starting from scratch, Viking and Saxon inherit from Soldier.
- This means they automatically get health, strength, attack(), and receiveDamage() from Soldier.
- They can then add their own unique features or change how existing features work.

Classes - Vikings

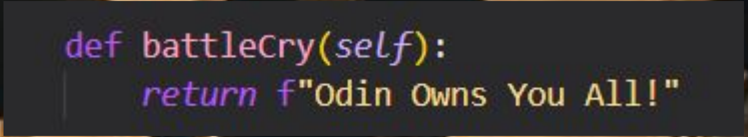


Viking:

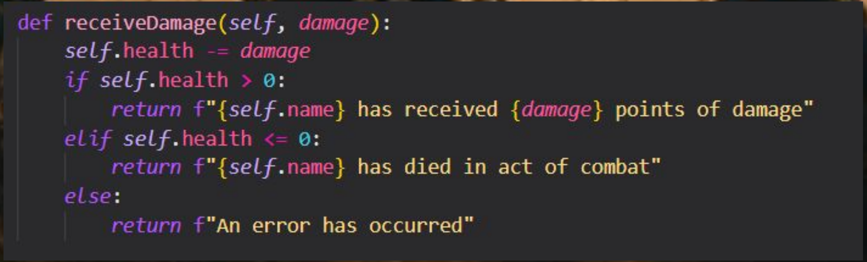
- Adds a name.
- Adds a battleCry() method.
- Custom message when receiveDamage() or dies.



```python
class Viking(Soldier):
    def __init__(self, name, health, strength):
        self.name = name
        super().__init__(health, strength)
```



```python
    def battleCry(self):
        return f"Odin Owns You All!"
```

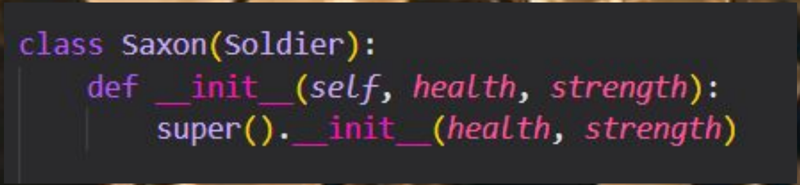


```python
def receiveDamage(self, damage):
    self.health -= damage
    if self.health > 0:
        return f"{self.name} has received {damage} points of damage"
    elif self.health <= 0:
        return f"{self.name} has died in act of combat"
    else:
        return f"An error has occurred"
```

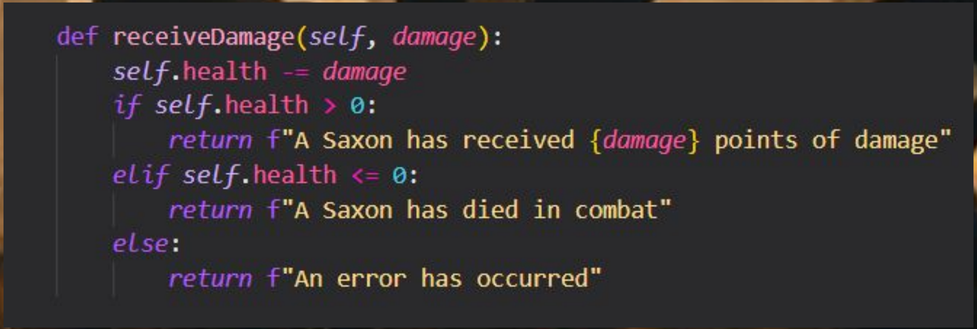
Classes - Saxons

Saxon:

- Doesn't receive a name
- Custom message when receiveDamage() or dies.



```python
class Saxon(Soldier):
    def __init__(self, health, strength):
        super().__init__(health, strength)
```

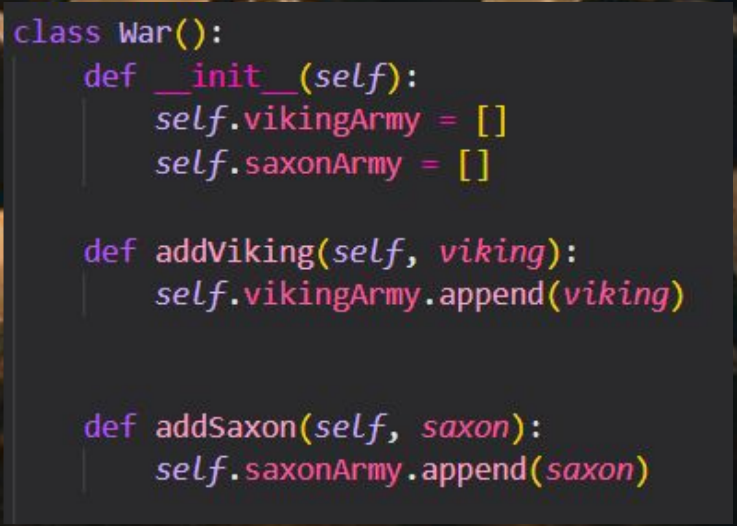


```python
def receiveDamage(self, damage):
    self.health -= damage
    if self.health > 0:
        return f"A Saxon has received {damage} points of damage"
    elif self.health <= 0:
        return f"A Saxon has died in combat"
    else:
        return f"An error has occurred"
```

The Battle Commander
The War Class

<u>Main Points:</u>

- This class acts as our "game master."
- What it HAS (Attributes):
  - vikingArmy: A list to hold all Viking objects.
  - saxonArmy: A list to hold all Saxon objects.

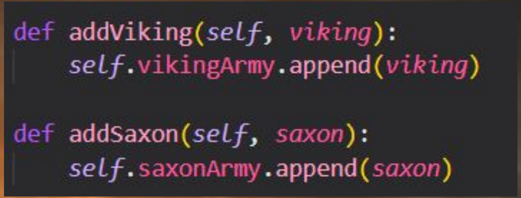


```python
class War():
    def __init__(self):
        self.vikingArmy = []
        self.saxonArmy = []

    def addViking(self, viking):
        self.vikingArmy.append(viking)


    def addSaxon(self, saxon):
        self.saxonArmy.append(saxon)
```

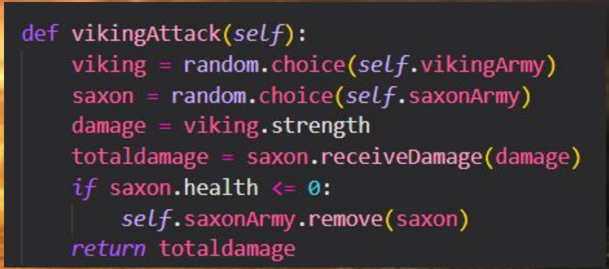
Simulating the Battle!

Time to place your bets…

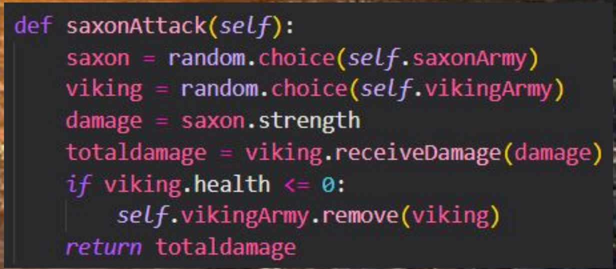
- What it CAN DO (Methods/Actions):
  - addViking() / addSaxon(): Recruit fighters to armies.
  - vikingAttack(): A random Viking attacks a random Saxon.
  - saxonAttack(): A random Saxon attacks a random Viking.
  - showStatus(): Tells us if the battle is ongoing, or who won!



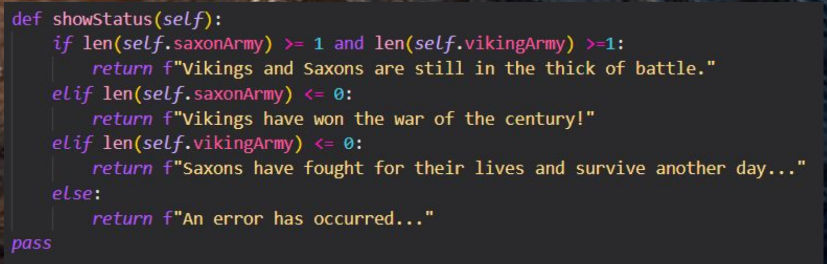
```python
def addViking(self, viking):
    self.vikingArmy.append(viking)

def addSaxon(self, saxon):
    self.saxonArmy.append(saxon)
```



```python
def vikingAttack(self):
    viking = random.choice(self.vikingArmy)
    saxon = random.choice(self.saxonArmy)
    damage = viking.strength
    totaldamage = saxon.receiveDamage(damage)
    if saxon.health <= 0:
        self.saxonArmy.remove(saxon)
    return totaldamage
```



```python
def saxonAttack(self):
    saxon = random.choice(self.saxonArmy)
    viking = random.choice(self.vikingArmy)
    damage = saxon.strength
    totaldamage = viking.receiveDamage(damage)
    if viking.health <= 0:
        self.vikingArmy.remove(viking)
    return totaldamage
```



```python
def showStatus(self):
    if len(self.saxonArmy) >= 1 and len(self.vikingArmy) >=1:
        return f"Vikings and Saxons are still in the thick of battle."
    elif len(self.saxonArmy) <= 0:
        return f"Vikings have won the war of the century!"
    elif len(self.vikingArmy) <= 0:
        return f"Saxons have fought for their lives and survive another day..."
    else:
        return f"An error has occurred..."
pass
```

Key Takeaways from Our Battle

- Organization: Classes help us structure our code, keeping fighter data and actions separate but connected.
- Reusability: We define Soldier once, and Viking and Saxon automatically get its features.
- Flexibility: We can easily add new types of fighters (e.g., "Archer," "Knight") without rewriting everything.
- Clearer Code: Easily understandable for English speakers (viking.attack(), saxon.receiveDamage()).
- Modeling Reality: Classes allow us to represent real-world concepts (like armies and soldiers) in our code.

Any Questions, warriors?