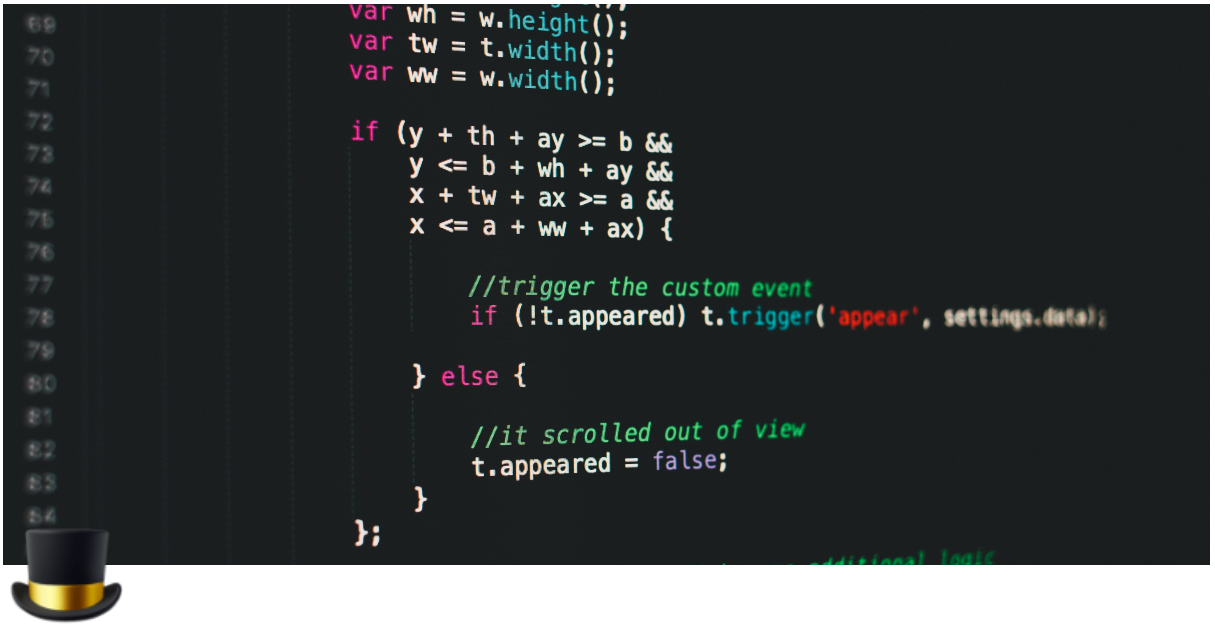

Intro to Object-Oriented Programming

▼ What is Object-Oriented Programming?

- OOP is a programming paradigm, meaning it is a way to program. A common feature is having classes with attributes attached to them, for example, a `Person` object has attributes `name` and `DOB`.
- OOP operates under concepts that include:
  - Classes and Objects
  - Inheritance
  - Polymorphism
  - and much more
- “ChatGPT, tell me in one sentence why programmers should use OOP.”
  - “Programmers should use Object-Oriented Programming (OOP) to better organize and structure code by encapsulating data and behavior into reusable and modular objects, leading to more maintainable and scalable software systems.”
  - In other words, OOP allows us to create organised, reusable pieces of code that manipulate things and actions by creating structures

called classes, in the process promoting encapsulation, abstraction, polymorphism and delegation.
    - Sounds too fancy...
- Some applications of OOP
    - Software Development, specifically for large industry projects
    - Game development (today's topic!)
- Most common languages that are object-oriented:
    1. Java
    2. C++
    3. Python
    4. C#
    5. JavaScript
    6. Ruby

Now, let's dive into the features of OOP...

▼ Classes & Objects

- A class is like a template, a blueprint for creating objects, things.
- For example, if you have a game
    - 1 player has a name and a super power (2 attributes)
    - Then 5 players have 10 attributes all together!
    - Picture this:

```
# Define their names
player1_name = "Eden"
player2_name = "Rama"
player3_name = "Kasie"
player4_name = "Kevin"
player5_name = "Prat"
```

```
# And their powers...
player1_power = "Gravity Manipulation"
player2_power = "Force Fields"
player3_power = "Super Speed"
player4_power = "Elemental control"
player5_power = "Flight"

print(player1_name + " has superpower: " + player1_power
print(player2_name + " has superpower: " + player2_power
print(player3_name + " has superpower: " + player3_power
print(player4_name + " has superpower: " + player4_power
print(player5_name + " has superpower: " + player5_power
```

    - Advantages? Easy to write, I guess... Not too much logical thinking going on
    - Disadvantages? To name a few...
        - Lack of abstraction, making it harder to maintain much larger databases.
        - Code duplication.
        - Limited reusability, making it extremely hard to modify or extend the code for different scenarios.
        - How do you know which player has which super powers?

- Solutions, here we go: Define a `Player` class that has 2 attributes, name and superpower:

```
class Player:
    def __init__(self, name, power):
        self.name = name
        self.power = power
```

- Brilliant! Now we want to create a new player, we do it like so:

```
player1 = Player("Eden", "Gravity Manipulation")
player2 = Player("Rama", "Force Fields")
```

```
player3 = Player("Kasie", "Super Speed")
# and so on... the previous are called objects, an object i
```

- But wait... what if I want to print the players? I'm still doing the same boring printing like before. Nope you are not! Let's add a functionality to our class, called a method!

```
class Player:
    def __init__(self, name, power):
        self.name = name
        self.power = power

    def describe(self):
        print(self.name + " has superpower: " + self.power)


# Create your objects, instances of your class
player1 = Player("Eden", "Gravity Manipulation")
player2 = Player("Rama", "Force Fields")
player3 = Player("Kasie", "Super Speed")


player1.describe()
player2.describe()
player3.describe()
```

▼ Inheritance

- Inheritance is when you have one class that inherits some or all of its properties from another class. We say the child class inherits from the parent class.
- For example, there are multiple characters in a game, some are heroes, and others are villains. So we define a parent class as follows:

```
class Character():
    def __init__(self, name, power):
        self.name = name
        self.power = power

    def describe(self):
        print(self.name, self.power)
```

- and from there, you can create a child class that has the same attributes and methods as those in the `Character` class.

```
# Parent class
class Character():
    def __init__(self, name, power):
        self.name = name
        self.power = power

    def describe(self):
        print(self.name, self.power)

# Child classes
class Hero(Character):
    def __init__(self, name, power, win_count):
        super().__init__(name, power)
        self.win_count = win_count

class Villain(Character):
    def __init__(self, name, power, story):
        super().__init__(name, power)
        self.story = story
```

- and so you have a common template for all the characters, but then each type of character has their own special attributes as well 🙂
- Now we can create objects just like we did earlier:

```
spiderman = Hero("Spiderman", "Spider Powers", 10)
DocOctopus = Villain("Doctor Octopus", "Electronic Arm
s", "Originally a brilliant scientist, his greatest inve
ntion, a set of metallic limbs, became fused to his body
by an accident which caused his insanity. He has telepat
hic control of these arms, which are strong enough to ph
ysically hurt Spider-Man.[12] While Doctor Octopus is re
garded as one of Spider-Man's archenemies, he also been
portrayed as an antihero, and even starred in his own co
mic book storyline that saw him becoming a superhero cal
led the Superior Spider-Man after the original Spider-Ma
n's death.")
```

▼ Polymorphism

- Means: many forms. It is like using the same function for different purposes.
- For example, take the following function

```
# len() function
print(len("Hello World!"))
print(len([1, 2, 3, 4, 5]))
print(len("Apples", "Bananas"))
print(len({"num1": 1, "num2": 2, "num3": 3}))
```

- Notice how the `len()` function can be used for finding the length of so many different python objects.
- This is a fantastic example of polymorphism. You don’t need to know what’s on the other side (what `len` is doing behind the scenes), but you know that it can take a string, list, tuple or dictionary as input, and it’ll throw the length back at you.
- How can that be done in our game? Take a look at the following:

```
# Parent class
class Character():
    def __init__(self, name, power):
        self.name = name
```

```
        self.power = power

    def describe(self):
        print(self.name, self.power)

    def move(self):
        print("Move!")

# Child classes
class Hero(Character):
    def __init__(self, name, power, win_count):
        super().__init__(name, power)
        self.win_count = win_count

    def move(self):
        print("Run!")

class Villain(Character):
    def __init__(self, name, power, story):
        super().__init__(name, power)
        self.story = story

    def move(self):
        print("Chase!")


spiderman = Hero("Spiderman", "Spider Powers", 10)
DocOctopus = Villain("Doctor Octopus", "Electronic Arm
s", "Originally a brilliant scientist, his greatest inve
ntion, a set of metallic limbs, became fused to his body
by an accident which caused his insanity. He has telepat
hic control of these arms, which are strong enough to ph
ysically hurt Spider-Man.[12] While Doctor Octopus is re
garded as one of Spider-Man's archenemies, he also been
portrayed as an antihero, and even starred in his own co
mic book storyline that saw him becoming a superhero cal
led the Superior Spider-Man after the original Spider-Ma
n's death.")
```

```
spiderman.move()
DocOctopus.move()
```

```
# Animal data structures
elephant = {"name": "Ellie", "species": "Elephant", "age": 10
lion = {"name": "Leo", "species": "Lion", "age": 5}

# Functions for interacting with animals
def introduce_animal(animal):
    print(f"This is {animal['name']}, a {animal['species']}."

def feed_animal(animal):
    print(f"Feeding {animal['name']} the {animal['species']}.

# Using functions
introduce_animal(elephant)
feed_animal(lion)
```

```
class Animal:
    def __init__(self, name, species, age):
        self.name = name
        self.species = species
        self.age = age

    def feed(self):
      print(f"Feeding {self.name} the {self.species}.")

  def introduce(self):
      print(f"This is {self.name}, a {self.species}.")
```

```
# Creating instances of the Animal class
lion = Animal("Leo", "Lion", 5)
elephant = Animal("Ellie", "Elephant", 10)

# Using methods
lion.feed()
elephant.introduce()
```

```
Feeding Leo the Lion.
This is Ellie, a Elephant.
```