Creative Coding 7

David Lynch

Slide Deck by Ruán Murgatroyd

Last Class

- Declarative and Imperative Programming.
- Introduction to *Recursion*.
- Recursive problem solving.
- Working with lists recursively.
- Recursive art in *Processing*.

Today's Class

- Intro to Object Oriented Programming (OOP)
- OOP's History
- Class inheritance
- Polymorphism
- Inner classes and functions
- Details on the Art Expo

Object Oriented Programming

Object Oriented Programming

OOP is a paradigm of imperative programming.

Everything is defined as an *Object*.

The most simple type of data is an object and every other type inherits from that.

OOP's History

The first OOP Language was *Smalltalk*, created by Alan Kay in 1969.

It is a style of programming structured around data where objects are data fields which have unique attributes and behaviours.

Another core OOP feature is that idea that objects can communicate with eachother.

Inheritance

Inheritance

Similar to how you inherit traits from your parents, classes inherit traits from other classes.

This allows us to extend class functionality when creating new ones.

This means we don’t have to repeat attributes over and over again.

Inheritance

inheritance.py

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

    def introduce(self):
        print("Hello, my name is:", self.name, "\nMy age is:", self.age)

class Dog(Animal):
    def __init__(self, name, age, breed):
        super().__init__(name, age)
        self.breed = breed

    def bark(self):
        self.introduce()
        print("Woof Woof Woof, I am a", self.breed)
```

`super()`

When one class inherits from another, that is called a *parent-child relationship*, where the child inherits from the parent.

When using inheritance, you access the parent's attributes with `super()`.

This is because we need to access the ***SUPERCLASS***.

Polymorphism

Polymorphism

Polymorphism means *Many Forms*. In programming, it means our classes can do the same thing in many ways.

i.e. Dogs, cats and ducks are all animals and have a call. Since they are different, our base `Animal` class in the next code snippet will have a `call` behaviour, but each sub-class will have a different call.

polymorphism.py

```python
class Animal:
    def call(self):
        pass
    def badInit(self):
        print("Bad initializer")
        self = None

class Monkey(Animal):
    def call(self):
        print("OO OO AA AA")

class Cat(Animal):
    def call(self):
        print("Meow")

class Dog(Animal):
    def call(self):
        print("WOOF")
```

Accessor Methods

Accessor methods allow us to control the way an object's attributes are accessed.

We use *getters* and *setters* to access our attributes instead of the typical *x.y* syntax.

This means that we can create rules for how we construct our classes and validate types and other items.

Encapsulation

Encapsulation is the concept of wrapping components into a class or other composite type.

We can take *Pong* as an example, we could have the paddles, ball, score, and game logic be held in separate locations, or we could wrap them all in a `Game` class to let us have finer control over how our game is run.

Warm-Up Questions

Warm-Up Questions

1. Create a class called `Human`. C
2. Give it human attribuandtes like `name` and `age`.
3. Write an `__init__` method for your human.
4. Write a method to allow your human to `speak` and `eat`.
5. Create a class for different types of people such as doctors, programmers, etc.
6. Write an `__init__` method for each type and use the `super()`
7. method to initialise their `name`, etc.
8. Get creative!

The End

Thanks for coming!

Notes uploaded on the Discord.

See you next week!