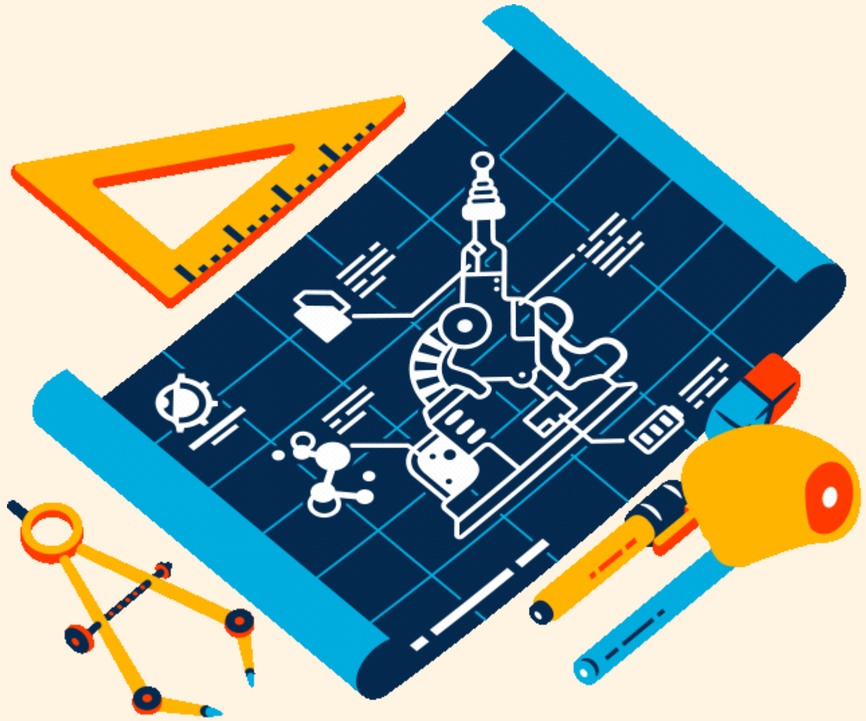
OOP



a way of organizing code

WTF is OOP

Object Oriented Programming is an approach to programming that involves modeling "things" into Python objects.

These objects can contain both **data AND functionality** all wrapped up together into a reusable component.

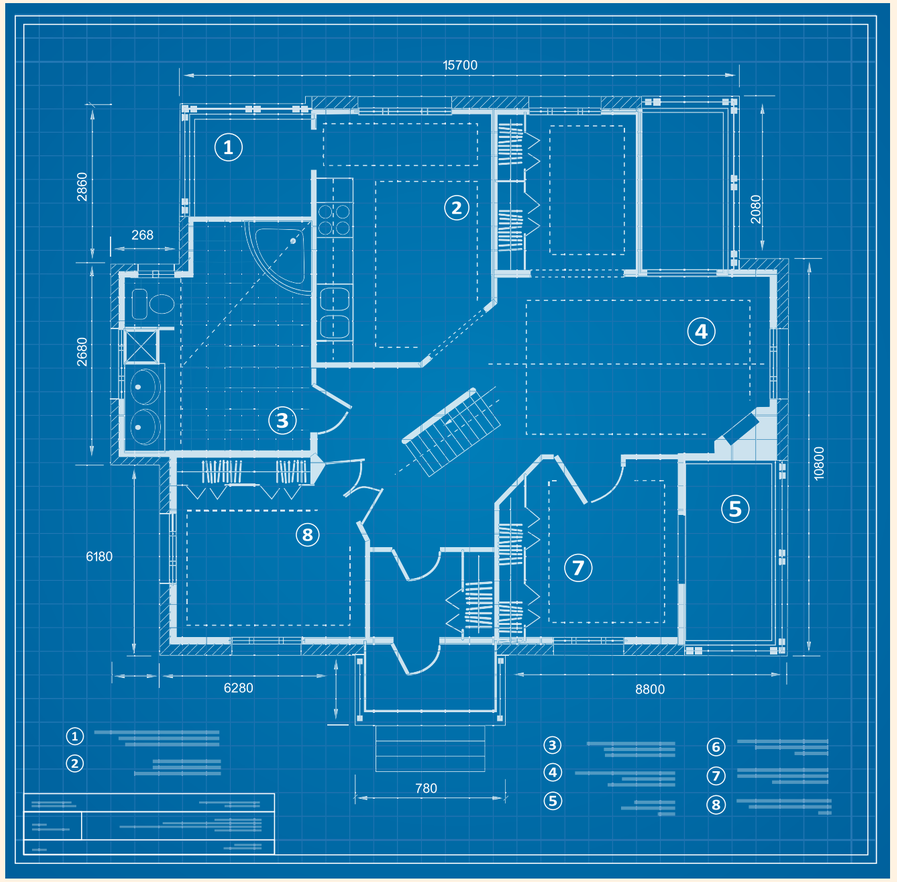
Chess

- Board
- Piece
- Player
- AIPlayer
- Match

Class

Classes are blueprints or recipes that we can later use to create objects from.

A class describes what properties and functionality individual objects will contain



Instance

We call the individual objects that are created from a class blueprint **instances.**

Class

Cupcake Recipe

- flour type
- flavor
- frosting
- color

Instances

Class

Cupcake Recipe

- flour type
- flavor
- frosting
- color

Instances

Class

Cupcake Recipe

- flour type
- flavor
- frosting
- color

Instances

Class

Instances

ChessPiece

Data

- color
- piece_type

Class

ChessPiece

Data

- color
- piece_type

Instances

- white
- knight

Class

ChessPiece

Data

- color
- piece_type

Instances

- white
- knight

- black
- pawn

Class

Instances

Player

Data

- name
- class
- inventory
- health

Functionality

- save()
- restart()
- heal()

Class

Player

Data

- name
- class
- inventory
- health

Functionality

- save()
- restart()
- heal()

Instances

- Titus
- Warrior
- ['sword']
- 81

Class

Player

Data

- name
- class
- inventory
- health

Functionality

- save()
- restart()
- heal()

Instances

- Titus
- Warrior
- ['sword']
- 81

- Albus
- Mage
- ['staff', 'hat']
- 100

Class

Task

Data

- name
- description
- status
- priority
- owner

Functionality

- assign()
- complete()

Instances

Class

Task

Data

- name
- description
- status
- priority
- owner

Functionality

- assign()
- complete()

Instances

- Bugfix
- Fix the double click bug on home page navbar
- Medium Priority
- Theresa

Class

Task

Data

- name
- description
- status
- priority
- owner

Functionality

- assign()
- complete()

Instances

- Bugfix
- Fix the double click bug on home page navbar
- Medium Priority
- Theresa

- Login Ticket
- Create new login screen for app
- High Priority
- Baker

Class

Deck

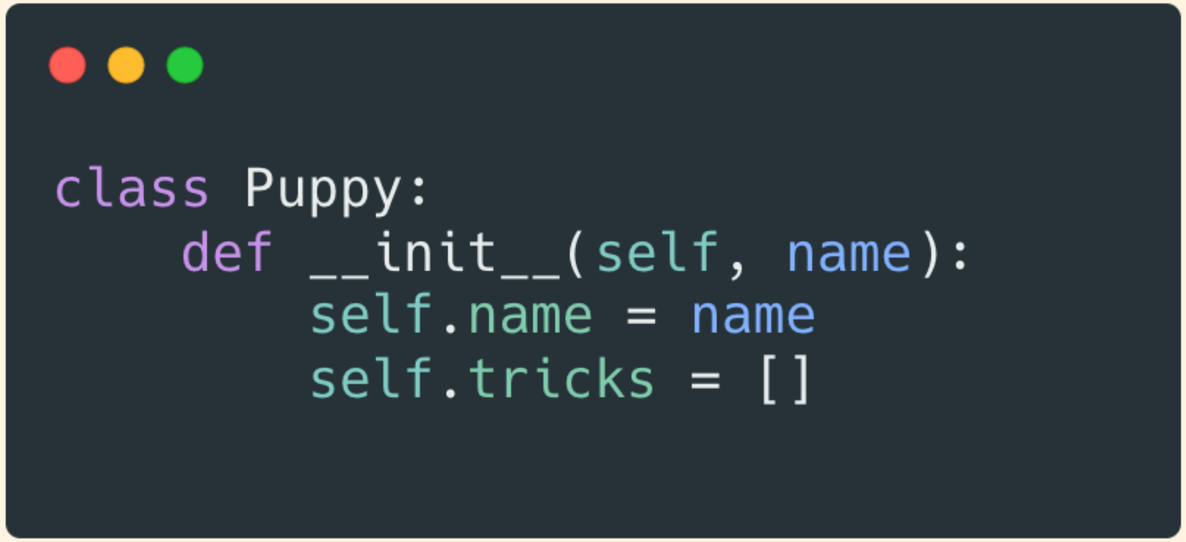
Data

- cards

Functionality

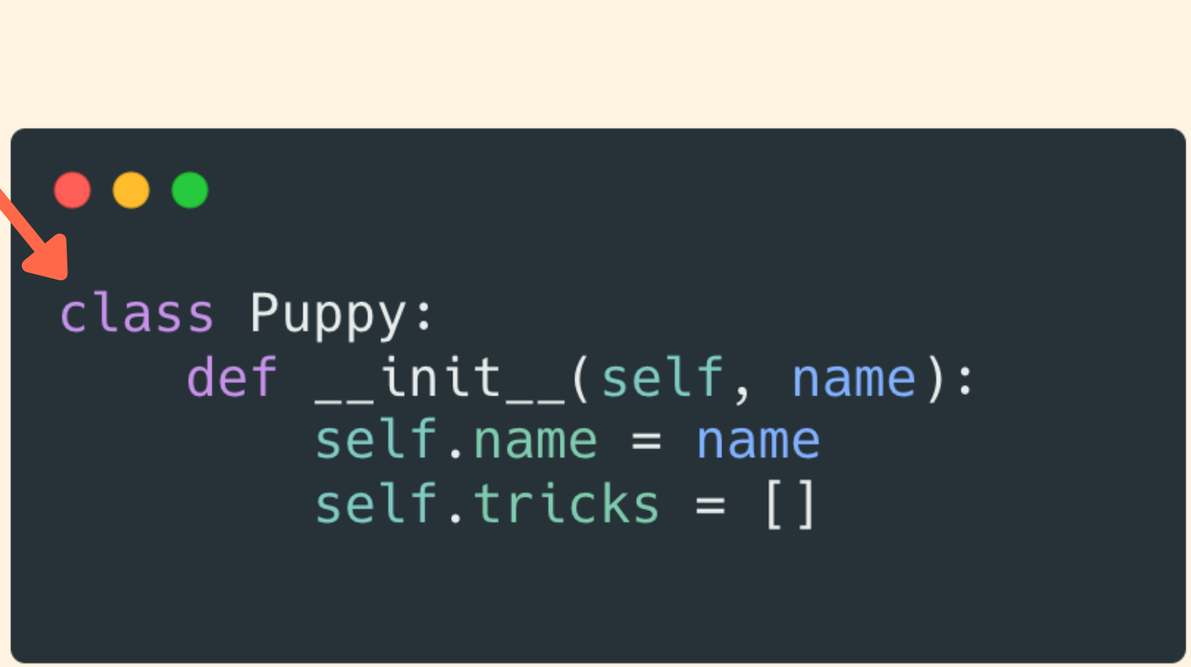
- shuffle()
- deal()

Instances



```
class Puppy:
    def __init__(self, name):
        self.name = name
        self.tricks = []
```

Class Keyword

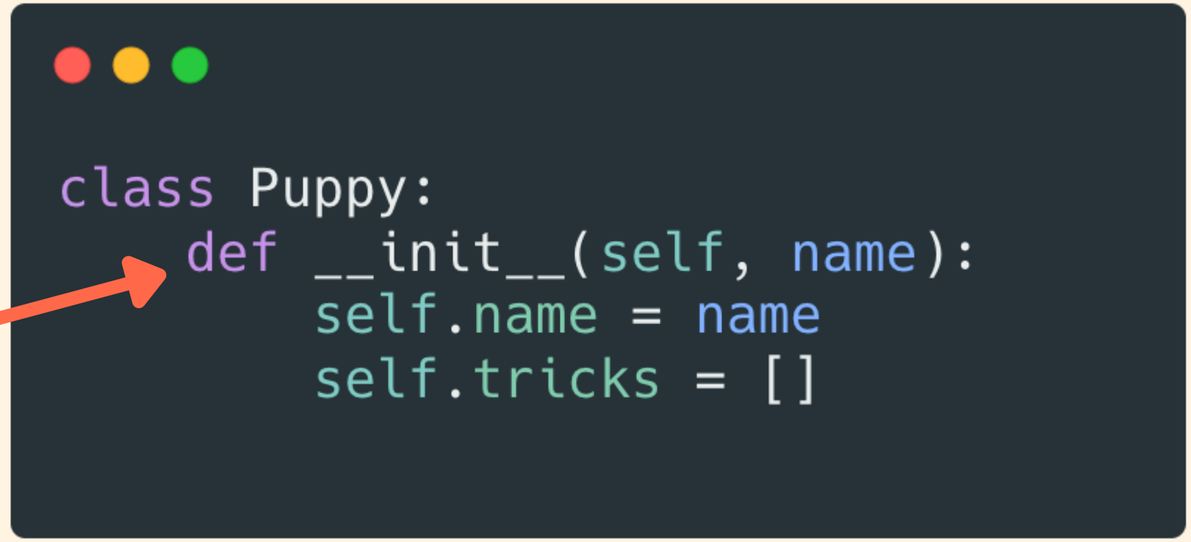


```
class Puppy:
    def __init__(self, name):
        self.name = name
        self.tricks = []
```

Class Name (Capitalized)

Special init method

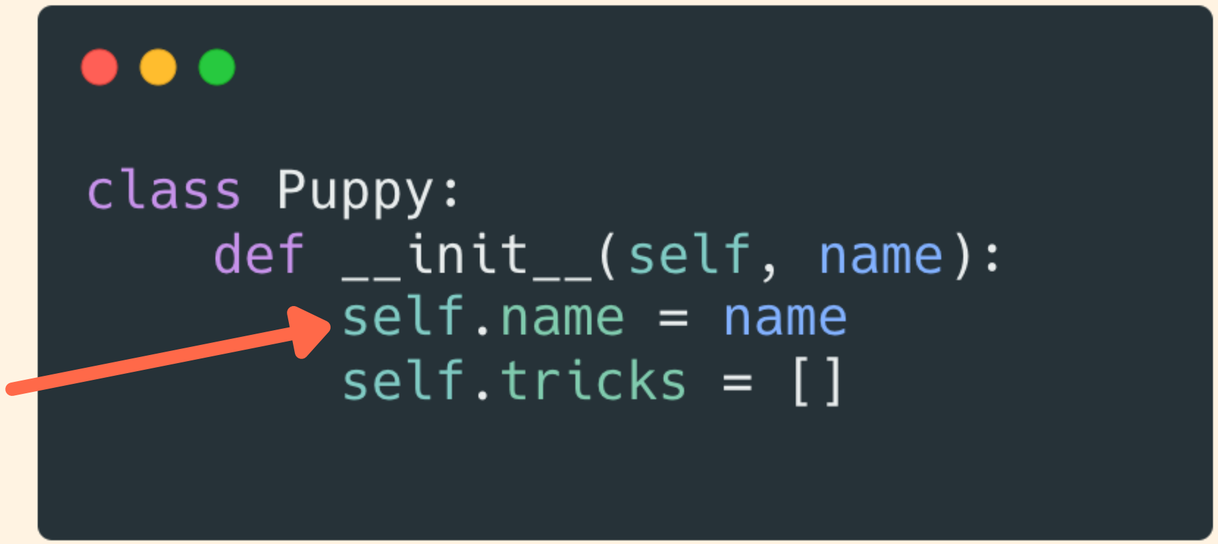
automatically called whenever new Puppy is created



```
class Puppy:
    def __init__(self, name):
        self.name = name
        self.tricks = []
```

self keyword

The self keyword refers to the "current" instance of Puppy



```
class Puppy:
    def __init__(self, name):
        self.name = name
        self.tricks = []
```

self keyword

self must be the first parameter to init

```
class Puppy:
    def __init__(self, name):
        self.name = name
        self.tricks = []
```

```python
class Puppy:
    def __init__(self, name):
        self.name = name
        self.tricks = []
```

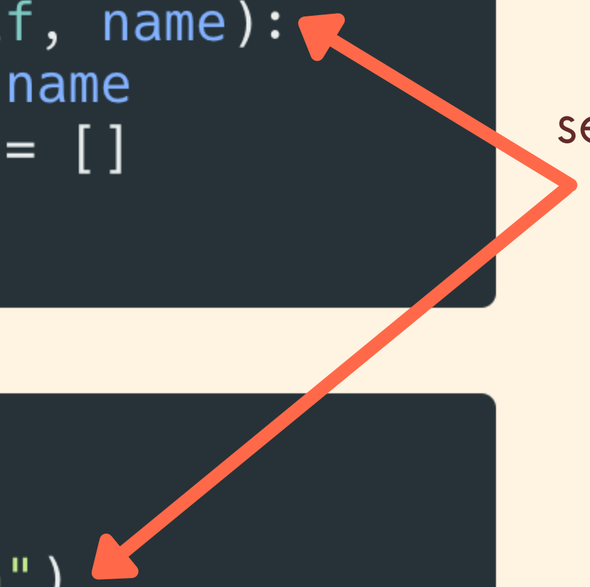
```python
elton = Puppy("Elton")
elton.name
'Elton'
elton.tricks
[]
```

instatiating

To create a Puppy instance, call Puppy() and provide a name

```
class Puppy:
    def __init__(self, name):
        self.name = name
        self.tricks = []
```

```
elton = Puppy("Elton")
elton.name
'Elton'
elton.tricks
[]
```



Init

The init method runs.
self.name is set to "Elton"
self.tricks is set to []

```python
class Puppy:
    def __init__(self, name):
        self.name = name
        self.tricks = []
```

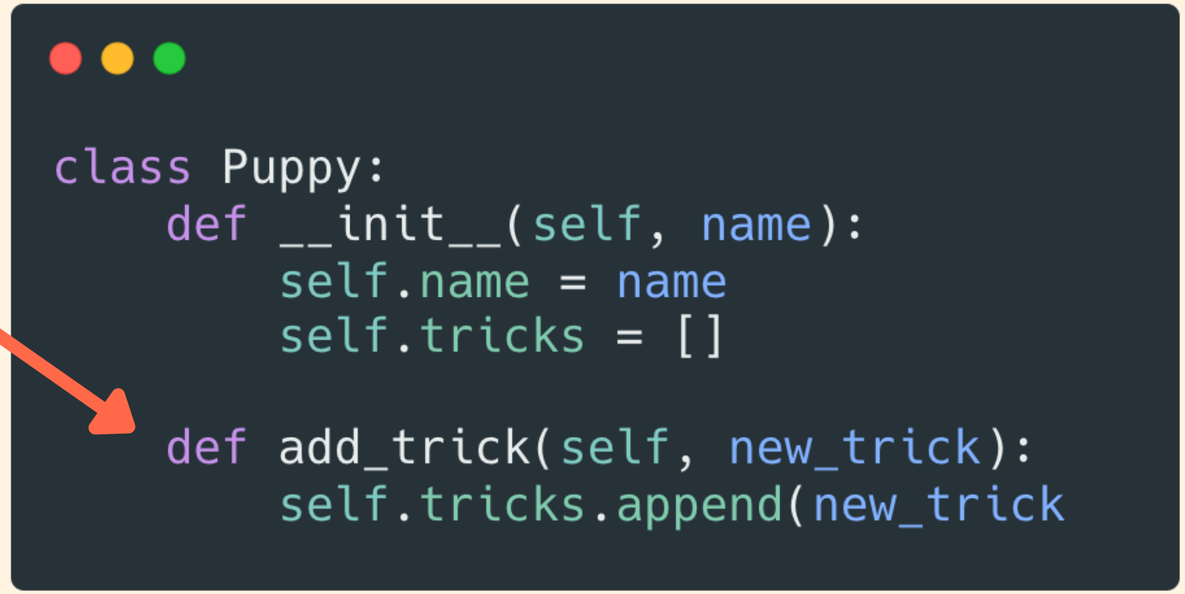
```python
otter = Puppy("Otter")
otter.name
'Otter'
otter.tricks
[]
```

Do it again!

Every time we call Puppy() we're creating a new Puppy instance with its own name and tricks list.

Add a method!

This add_trick() method appends a new trick to a Puppy instance's tricks list



```
class Puppy:
    def __init__(self, name):
        self.name = name
        self.tricks = []

    def add_trick(self, new_trick):
        self.tricks.append(new_trick
```

```
class Puppy:
    def __init__(self, name):
        self.name = name
        self.tricks = []

    def add_trick(self, new_trick):
        self.tricks.append(new_trick
```

Call it on an instance!

calling add_trick() on elton adds 'sit' to his tricks list

```
elton = Puppy("Elton")
elton.tricks
[]
elton.add_trick('sit')
elton.tricks
['sit']
```

add another instance method

perform_trick will perform a trick if a Puppy instance knows the trick.

```
class Puppy:
    def __init__(self, name):
        self.name = name
        self.tricks = []

    def add_trick(self, new_trick):
        self.tricks.append(new_trick)

    def perform_trick(self, trick):
        if trick in self.tricks:
            print(f"{self.name} performs {trick}!")
        else:
            print(f"{self.name} doesn't know {trick}"
```

Each Puppy instance is a different object

Elton knows sit but not stay

```
elton = Puppy('elton')
elton.add_trick('sit')

elton.perform_trick('sit')
"elton performs sit!"

elton.perform_trick('stay')
"elton doesn't know stay"
```

```
lando = Puppy('Lando')
lando.add_trick('sit')
lando.add_trick('down')
lando.add_trick('stay')

lando.perform_trick('stay')
"Lando performs stay!"
```

Lando is a totally separate Puppy with his own tricks list

Lando knows sit, down, and stay

```
elton = Puppy('elton')
elton.add_trick('sit')

elton.perform_trick('sit')
"elton performs sit!"

elton.perform_trick('stay')
"elton doesn't know stay"
```

```
lando = Puppy('Lando')
lando.add_trick('sit')
lando.add_trick('down')
lando.add_trick('stay')

lando.perform_trick('stay')
"Lando performs stay!"
```

Class Attributes

species is defined on the class itself

All instances of Puppy will have the same value for species

```
class Puppy:

    species = 'canine'

    def __init__(self, name):
        self.name = name
        self.tricks = []
```

```
class Puppy:

    species = 'canine'

    def __init__(self, name):
        self.name = name
        self.tricks = []
```

All instances of Puppy will have the same value for species

```
bozo = Puppy('Bozo')
enya = Puppy('Enya')

bozo.species
'canine'

enya.species
'canine'
```

Class Methods

```
class Puppy:
    species = 'canine'

    @classmethod
    def register_anon(cls):
        return cls('unknown')

    def __init__(self, name):
        self.name = name
        self.tricks = []
```

We can define methods that are available on the class directly. These class methods are not concerned with a specific instance of the class.

Class Methods

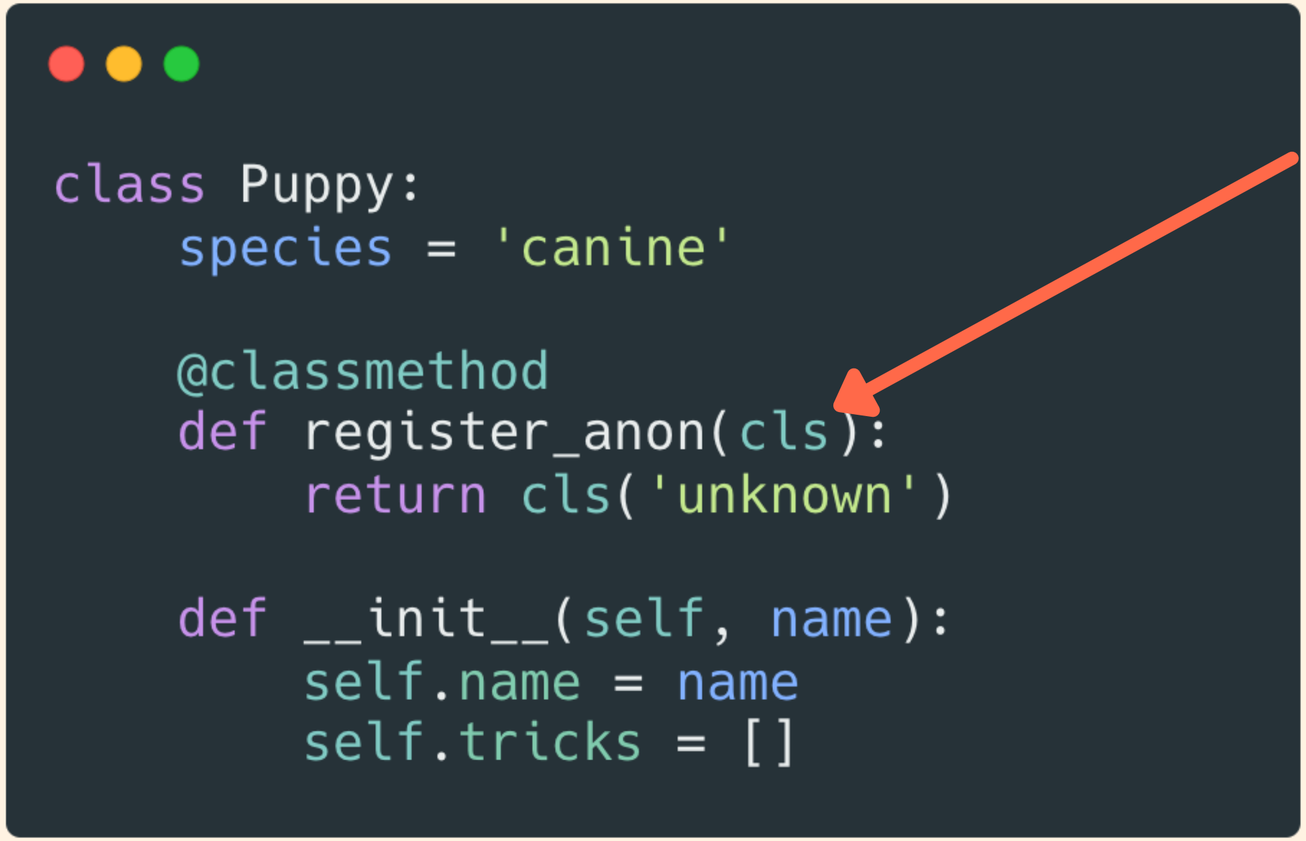
We use the **@classmethod** decorator

```
class Puppy:
    species = 'canine'

    @classmethod
    def register_anon(cls):
        return cls('unknown')

    def __init__(self, name):
        self.name = name
        self.tricks = []
```

Class Methods



```
class Puppy:
    species = 'canine'

    @classmethod
    def register_anon(cls):
        return cls('unknown')

    def __init__(self, name):
        self.name = name
        self.tricks = []
```

use **cls** as the first parameter. It will refer to the class itself

Class Methods

```
class Puppy:
    species = 'canine'

    @classmethod
    def register_anon(cls):
        return cls('unknown')

    def __init__(self, name):
        self.name = name
        self.tricks = []
```

```
stray = Puppy.register_anon()
```

We can call the method directly on the class!

Inheritance

```python
class Cat:
    def __init__(self, name):
        self.name = name
    def meow(self):
        print(f"{self.name} meows")
```

```python
class Lion(Cat):
    def roar(self):
        print(f"{self.name} roars")
```

Inheritance

```python
class Cat:
    def __init__(self, name):
        self.name = name
    def meow(self):
        print(f"{self.name} meows")
```

```python
class Lion(Cat):
    def roar(self):
        print(f"{self.name} roars")
```

```python
eli = Lion('Eli')
eli.meow()
"Eli meows"
```

eli is a Lion, but he can meow because Lion inherits from Cat.

Super()

use **super()** to refer to the base or parent class.

In this case, we use super() to access the __init__ method on from the Cat class.

```python
class Cat:
    def __init__(self, name):
        self.name = name
    def meow(self):
        print(f"{self.name} meows")
```

```python
class Lion(Cat):

    def __init__(self, name, pride_name):
        super().__init__(name)
        self.pride_name = pride_name

    def roar(self):
        print(f"{self.name} roars")
```