INTRO2CS

TIRGUL 9 - OOP

Overview

Today is all about OOP!

We'll see:

- Classes
- Objects
- self
- Methods
- Constructor
- Member/Class Variables
- Private Members/Methods
- Encapsulation (API)

Introduction

So far, our programs were made of different variables and functions operating on them.

This programming paradigm is called **Procedural Oriented Programming**.

Object Oriented Programming is a different programming paradigm!

Introduction

OOP breaks the programming task into objects, which combine data (known as attributes) and behaviors/functions (known as methods).

POP vs OOP

PROCEDURAL ORIENTED PROGRAMMING

○ Variables

○ Functions

OBJECT ORIENTED PROGRAMMING

○ Objects

○ Methods

POP vs OOP

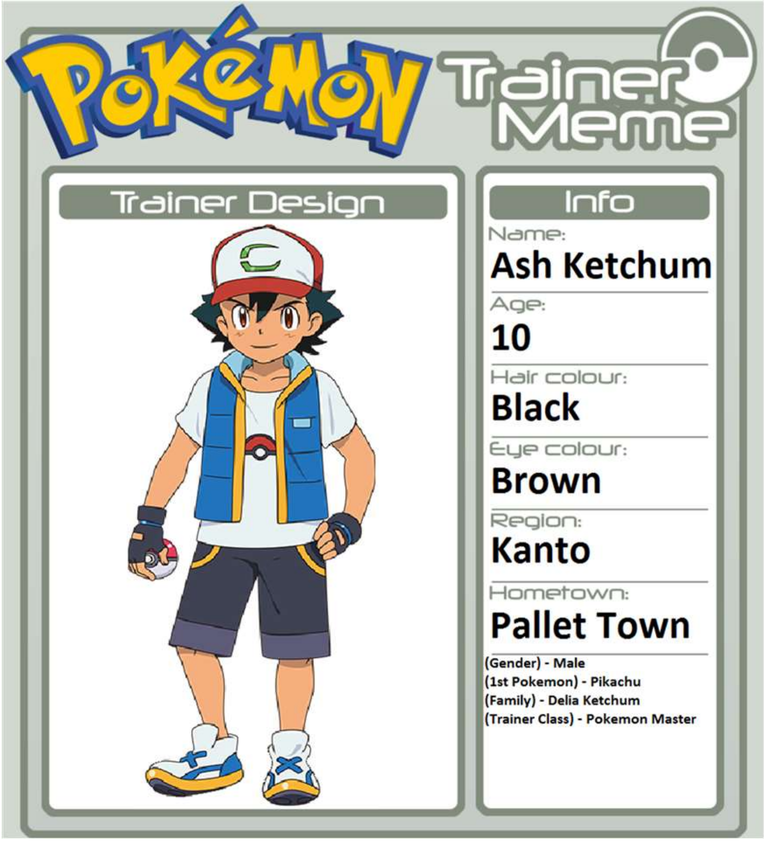
- Python is a multi-paradigm programming language
  - You do not have to use **OOP** when programming in Python (unlike other languages)
- You can still write very powerful programs using **POP**
- That said, **POP** is good for simple and small programs, while **OOP** is better suited for large programs

Let us take a closer look at **OOP**

Let's Play Pokémon!

Ash Ketchum

```
name = 'Ash Ketchum'

age = 10

home_town = 'Pallet Town'

trainer_class = 'Pokemon Master'
```



Name:
Ash Ketchum

Age:
10

Hair colour:
Black

Eye colour:
Brown

Region:
Kanto

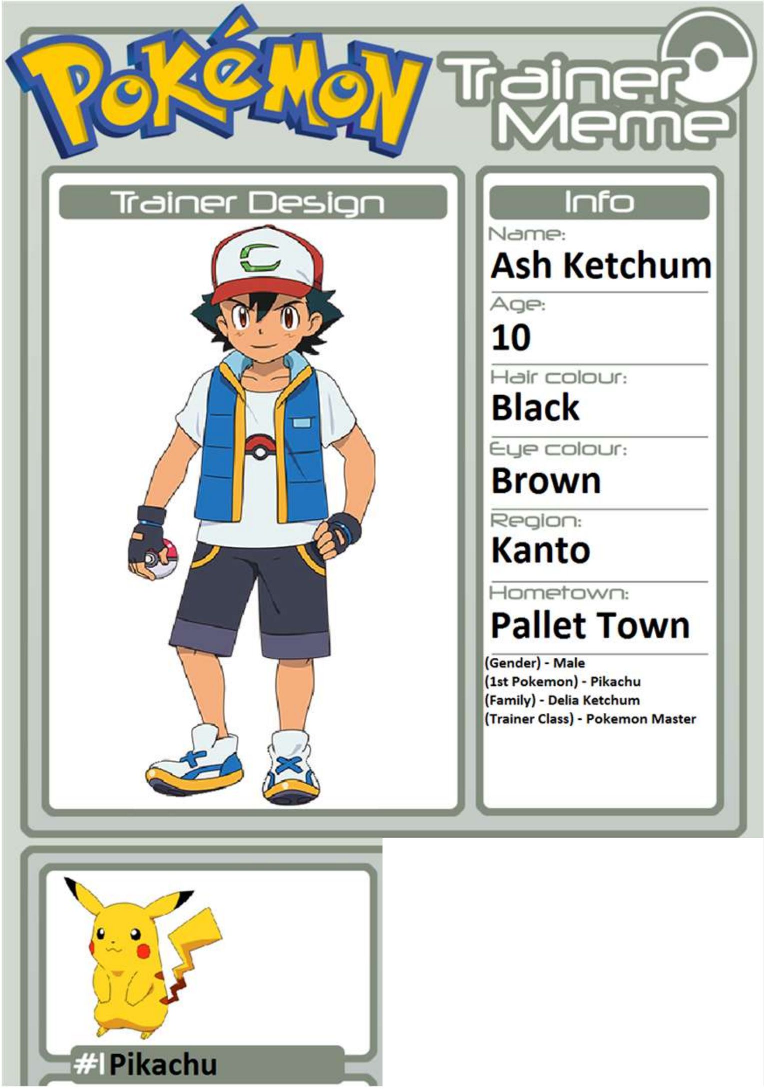
Hometown:
Pallet Town

(Gender) - Male
(1st Pokemon) - Pikachu
(Family) - Delia Ketchum
(Trainer Class) - Pokemon Master

What about Pokémons?

Add Pikachu

```
name = 'Ash Ketchum'

age = 10

home_town = 'Pallet Town'

trainer_class = 'Pokemon Master'


pokemon_name = 'Pikachu'

pokemon_type = 'Electric'

pokemon_level = 12

pokemon_attacks = ['Thunderbolt', 'Quick Attack', ...]
```



Info

Name:
Ash Ketchum

Age:
10

Hair colour:
Black

Eye colour:
Brown

Region:
Kanto

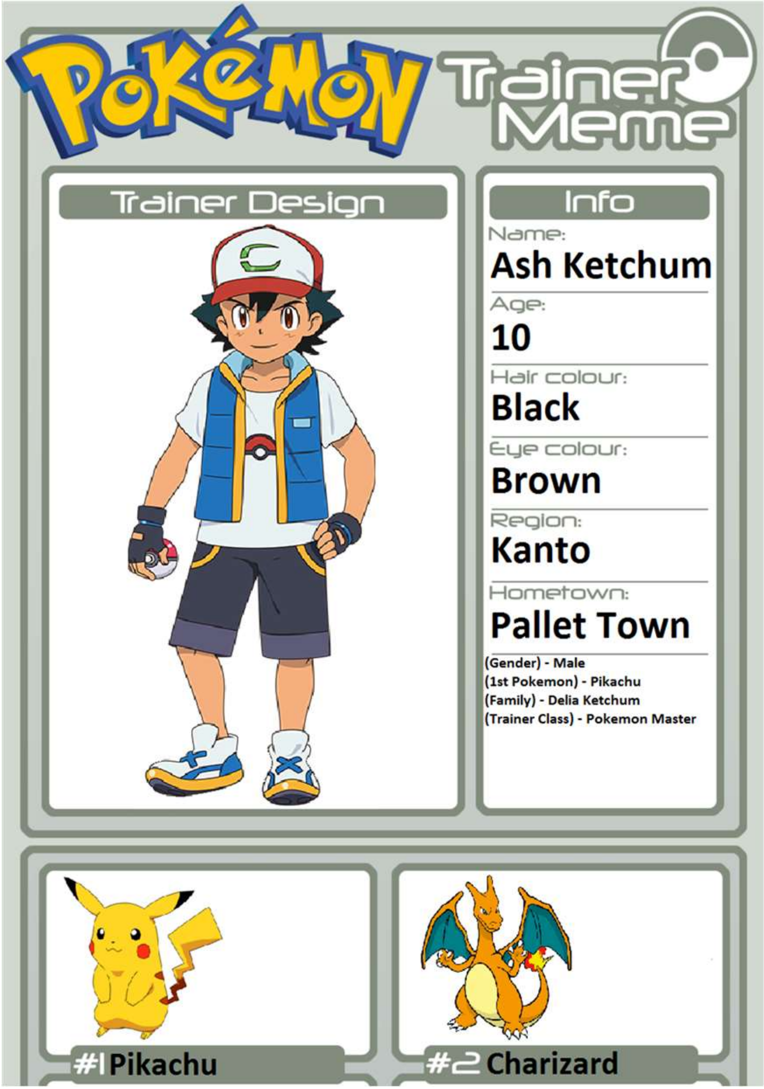
Hometown:
Pallet Town

(Gender) - Male
(1st Pokemon) - Pikachu
(Family) - Delia Ketchum
(Trainer Class) - Pokemon Master

#1 Pikachu

Add Charizard

```
name = 'Ash Ketchum'
age = 10
home_town = 'Pallet Town'
trainer_class = 'Pokemon Master'

pokemon_name = 'Pikachu'
pokemon_type = 'Electric'
pokemon_level = 12
pokemon_attacks = ['Thunderbolt', 'Quick Attack', …]

pokemon_2_name = 'Charizard'
pokemon_2_types = ['Fire', 'Flying']
pokemon_2_level = 12
pokemon_2_attacks = ['Flamethrower', 'Dragon Claw', …]
```



Name:
Ash Ketchum

Age:
10

Hair colour:
Black

Eye colour:
Brown

Region:
Kanto

Hometown:
Pallet Town

(Gender) - Male
(1st Pokemon) - Pikachu
(Family) - Delia Ketchum
(Trainer Class) - Pokemon Master

#1 Pikachu

#2 Charizard

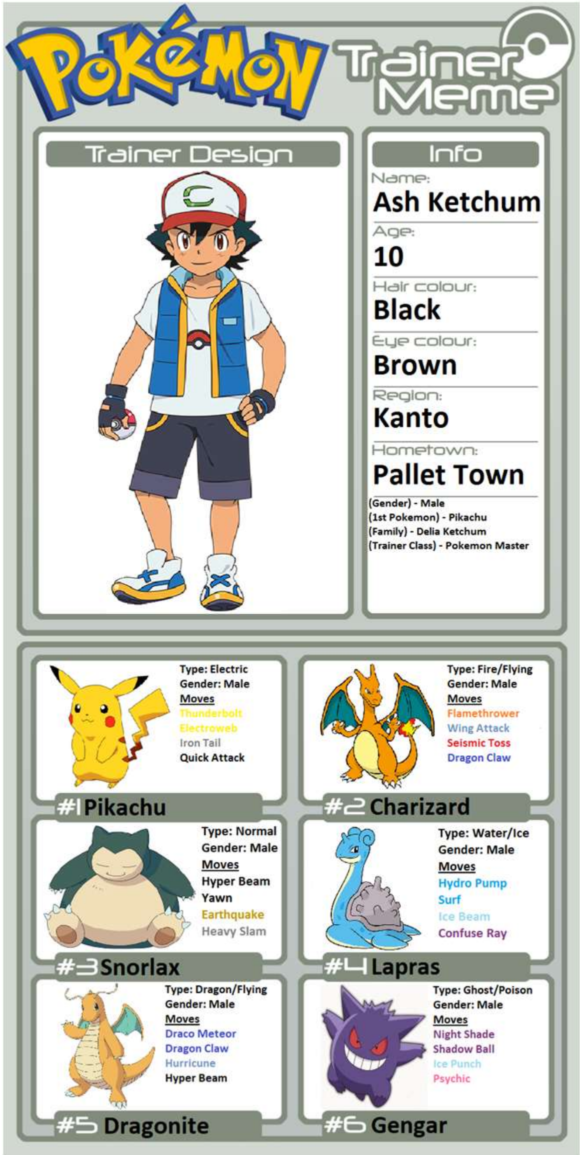
Another Solution?

```
name = 'Ash Ketchum'
age = 10
home_town = 'Pallet Town'
trainer_class = 'Pokemon Master'

pokemon_1 = {'name': 'Pikachu', 'type': ['Electric'], 'level': 12, 'attacks': ['Thunderbolt', 'Quick Attack', …]}
pokemon_2 = {'name': 'Charizard', 'type': ['Fire', 'Flying'], 'level': 27, 'attacks': ['Flamethrower', 'Dragon Claw', …]}
```

How About This?

```
name = 'Ash Ketchum'

age = 10

home_town = 'Pallet Town'

trainer_class = 'Pokemon Master'

pokemons = [

        {'name': 'Pikachu', 'type': ['Electric'], 'level': 12,

        'attacks': ['Thunderbolt', 'Quick Attack', …]},

        {'name': 'Charizard', 'type': ['Fire', 'Flying'], 'level': 27,

        'attacks': ['Flamethrower', 'Dragon Claw', …]},

        {'name': 'Snorlex’, …}…]
```

This is becoming complicated…



Info

Name:
Ash Ketchum

Age:
10

Hair colour:
Black

Eye colour:
Brown

Region:
Kanto

Hometown:
Pallet Town

(Gender) - Male
(1st Pokemon) - Pikachu
(Family) - Delia Ketchum
(Trainer Class) - Pokemon Master

#1 Pikachu
Type: Electric
Gender: Male
Moves
Thunderbolt
Electroweb
Iron Tail
Quick Attack

#2 Charizard
Type: Fire/Flying
Gender: Male
Moves
Flamethrower
Wing Attack
Seismic Toss
Dragon Claw

#3 Snorlax
Type: Normal
Gender: Male
Moves
Hyper Beam
Yawn
Earthquake
Heavy Slam

#4 Lapras
Type: Water/Ice
Gender: Male
Moves
Hydro Pump
Surf
Ice Beam
Confuse Ray

#5 Dragonite
Type: Dragon/Flying
Gender: Male
Moves
Draco Meteor
Dragon Claw
Hurricune
Hyper Beam

#6 Gengar
Type: Ghost/Poison
Gender: Male
Moves
Night Shade
Shadow Ball
Ice Punch
Psychic

What is the Solution?

The list of Pokémons will need to have a very complex structure.

Using **OOP** makes this much simpler!

The OOP Way – Classes

- The **class** is a blueprint to define a logical grouping of data and functions
- It provides a way to create data structures that model real-world entities

The Pokémon Class

For example, we can create a **Pokemon** class that contains:

- **Properties** such as name, type, and level
- **Behaviors** such as different attacks

Defining a Class

```python
class Pokemon:

    # class implantation
```

The OOP Way – Objects

- While **class** is the blueprint, an **object** is an **instance** of the class with actual values
- For example, a Pokémon named ‘Pikachu’ of type ‘Electric’.
- Put it another way – a **class** is like a template to define the needed information, and an **object** is one specific copy that filled in the template
- **Objects** created from the same **class** are independent from each other

Creating an Object

```
class Pokemon:
    # class implantation
```

```
my_first_pokemon = Pokemon()
print(type(my_first_pokemon))
```

The OOP Way – Members

- The properties that a **class** defines are called **data members** or **member variable**

- These **members** can have different values for different **objects**

- Accessing an **members** of an **object** is done in the following way: **object.member**

Setting Members

```
my_first_pokemon = Pokemon()

my_first_pokemon.name = 'Pikachu’

my_first_pokemon.type = 'Electric’

my_first_pokemon.level = 12
```

Having Many Pokémons

```
pokemons = []
```

Seems a bit inefficient

The OOP Way – Methods

- A **method** is a **class** function, meaning it is defined inside the **class** and can be called by **objects** of that **class**
- Calling a **method** is also done using a period: **object.method()**

Setting Members – Attempt 2

```
class Pokemon:
    def set_attributes(self, name, pokemon_type, level):
        self.name = name
        self.type = pokemon_type
        self.level = level
```

The Special Word self

- Methods are called by **specific** objects
- The first parameter of all methods is self
- self is the specific object that called the method
- We don’t need to pass this argument explicitly

Having Many Pokémons – Attempt 2

```
pokemons = []

my_first_pokemon = Pokemon()
my_first_pokemon.set_attributes('Pikachu', ['Electric'], 12)
pokemons.append(my_first_pokemon)

my_second_pokemon = Pokemon()
my_second_pokemon.set_attributes('Charizard', ['Fire', 'Flying'], 27)
pokemons.append(my_second_pokemon)

my_third_pokemon = Pokemon()
my_third_pokemon.set_attributes('Snorlex', ['Normal'], 6)
pokemons.append(my_third_pokemon)
```

Do we ever want to create a Pokémon without these attributes?

The Special Method __init__()

- Used for **initializing** the instances when we create the object of a class
- Also called a **constructor**
- All instances will have the members defined in the **constructor**

The Special Method __init__()

```
class Pokemon:
    def __init__(self, name, pokemon_type, level):
        self.name = name
        self.type = pokemon_type
        self.level = level
```

__init__ () is called immediately when we create a new **object**

```
my_pokemon = Pokemon('Pikachu', ['Electric'], 12)
```

The Special Method __init__()

- If no explicit **constructor** is defined →
  - default empty **constructor**:

```
class Pokemon:
    def __init__(self):
        # does nothing
```

- The **constructor** is always called __init__()

Having Many Pokémons – Attempt 3

```
pokemons = []

my_first_pokemon = Pokemon('Pikachu', ['Electric'], 12)
pokemons.append(my_first_pokemon)

my_second_pokemon = Pokemon('Charizard', ['Fire', 'Flying'], 27)
pokemons.append(my_second_pokemon)

my_third_pokemon = Pokemon('Snorlex', ['Normal'], 6)
pokemons.append(my_third_pokemon)
```

What Else Can We Do?

Eventually we want to battle. So, our Pokémons will need attributes such ah health, and they can be attacked and hurt. How can we implement this?

```
pokemon_1.health -= 10
if pokemon_1.health < 0:
    print(pokemon_1.name, 'fainted!')
```

Problematic?

Can lead to code duplication..

The non-OOP solution – use a function

```
def take_damage(pokemon, damage):
    pokemon.health -= damage
    if pokemon.health < 0:
        print(pokemon.name, 'fainted!')
```

Much better! But still, difficult to maintain. There may be many functions!

Using Methods

```
class Pokemon:

    def __init__(self, name, pokemon_type, level, health=50):
        self.name = name
        self.type = pokemon_type
        self.level = level
        self.health = health


    def take_damage(self, damage):

        self.health -= damage

        if self.health < 0:

            print(self.name, 'fainted!’)
```

```
my_pokemon = Pokemon('Pikachu', ['Electric'], 12)
my_pokemon.take_damage(10)
```

Class Variables

- In addition to **members** a **class** only defines, it can also store variables with values set in advance:

```
class Pokemon:
    num_pokemons = 898
    is_better_than_digimon = True

    def __init__(self, name, pokemon_type, level):
        # some implementation

    def take_damage(self, damage):
        # some implementation
```

Class Variables

- Class variables can be accessed and set by the **class**:

  ✓ Pokemon.num_pokemons

  ✓ Pokemon.num_pokemons = 899

  - When a class variable is set it also changes across all existing objects

- They can also be accessed by **objects** of the **class**, but cannot be set by them:

  ✓ my_pokemon.num_pokemons

  ✗ my_pokemon.num_pokemons = 899

- If we try to set a class variable via an instance:
  - we create a new member variable instead
  - The member variable will hide the class variable

Class Variables – Example

```
class Pokemon:
    num_pokemons = 898
    is_better_than_digimon = True

    # class implementation
```

```
my_pokemon = Pokemon('Pikachu', ['Electric'], 12)
print(my_pokemon.num_pokemons)
>> 898

Pokemon.num_pokemons += 1
print(my_pokemon.num_pokemons)
>> 899
```

Let’s Add a Class

```
class PokemonTrainer:
    def __init__(self, name, age, home_town, trainer_class):
        self.name = name
        self.age = age
        self.home_town = home_town
        self.trainer_class = trainer_class

 def attack(self, pokemon):
        pokemon.attack()
```

```
my_trainer = PokemonTrainer('Ash Ketcum', 10, 'Pallet Town', 'Pokemon Master')
my_pokemon = Pokemon('Pikachu', ['Electric'], 12)
my_trainer.attack(my_pokemon)
```

In each attack we need to provide our trainer a Pokémon, but the trainer owns his own Pokémons!

Let’s Add a Class

```
class PokemonTrainer:
  def __init__(self, name, age, home_town, trainer_class, pokemons):
    self.name = name
    self.age = age
    self.home_town = home_town
    self.trainer_class = trainer_class
    self.pokemons = pokemons

  def attack(self):
    self.pokemons[1].attack()
```

We can use Pokémon objects as members of a Pokémon trainer!

Safety First

Team Rocket gained access to your favorite Pokémon!

In Python, every method and member can be accessed **directly**!

So, our enemies can easily preform:

```
my_favorite_pokemon.level = 1
```

Private Members

In order to declare a **private** member, we use the following name convention:

```
class Pokemon:
  # some class variables
  def __init__(self, name, pokemon_type, level, health=50):
    self.__name = name
    self .__type = pokemon_type
    self .__level = level
    self .__health = health
```

Private Members – How Does it Work?

```
class Pokemon:
  # some class variables
  def __init__(self, name, pokemon_type, level, health=50):
    self.__name = name
    self .__type = pokemon_type
    self .__level = level
    self .__health = health
```

```
my_pokemon = Pokemon('Pikachu', ['Electric'], 12)
my_pokemon.__level -= 10
```

```
>> AttributeError: 'Pokemon' object has no attribute '__level'
```

Private Members – How Does it Work?

- Under the hood, Python changes the name of every member with a double underscore to:

```
object._class__variable
```

- This will work:

```
my_pokemon._Pokemon__level -= 10
```

But don’t do it.

When Direct Accesses is Denied

We use **getters** and **setters**

```
class Pokemon:
    # some class variables
    def __init__(self, name, pokemon_type, level, health=50):
        self.__name = name
        self .__type = pokemon_type
        self .__level = level
        self .__health = health

    def get_level(self):
        return self.__level

    def set_level(self, new_level):
        self.__level = new_level
```

Private Methods – Works the Same

```
class Pokemon:
    # some class variables
    def __init__(self, name, pokemon_type, level, health=50):
        self.__name = name
        self .__type = pokemon_type
        self .__level = level
        self .__health = health

    def __attack(self):
        # some attack
```

```
my_pokemon = Pokemon('Pikachu', ['Electric'], 12)
my_pokemon.__attack()

>> AttributeError: 'Pokemon' object has no attribute '__attack'
```

Encapsulation

In OOP languages, **encapsulation** is used to refer to one of the following:

- ○ A language mechanism for restricting direct access to some of the **object**'s components.
- ○ A language construct that facilitates the bundling of data with the **methods** (or other functions) operating on that data

Why Encapsulation?

- Prevent bugs (like changing a data member)
  - Using private members
- Hide specific implementations – a Pokemon doesn’t need to know how the trainer is implemented
  - **API**

Application Programming Interface

- The **API** defines the functionality an **object** allows
- Explains how to interact with an **instance**
- “Design by Contract”
- Not committed to internal implementation!

Everything in Python is an Object!

Now we finally understand the syntax that uses a period!

```
lst = [1, 2, 3]
lst.append(4)
''.join(lst)
```

Everything in Python is an Object!

The function dir() allows us to view all the attributes (member and methods) of an object

```
print(dir(lst))
>> ['__add__', '__class__', '__class_getitem__', '__contains__', '__delattr__', '__delitem__', '__dir__', '__doc__',
'__eq__', '__format__', '__ge__', '__getattribute__', '__getitem__', '__gt__', '__hash__', '__iadd__', '__imul__',
'__init__', '__init_subclass__', '__iter__', '__le__', '__len__', '__lt__', '__mul__', '__ne__', '__new__',
'__reduce__', '__reduce_ex__', '__repr__', '__reversed__', '__rmul__', '__setattr__', '__setitem__', '__sizeof__',
'__str__', '__subclasshook__', 'append', 'clear', 'copy', 'count', 'extend', 'index', 'insert', 'pop', 'remove', 'reverse',
'sort']
```

Special Methods

```
print(dir(my_pokemon))
>> ['_Pokemon__atack', '_Pokemon__health’, '_Pokemon__level', '_Pokemon__name', '_Pokemon__type',
'__class__', '__delattr__', '__dict__', '__dir__', '__doc__', '__eq__', '__format__', '__ge__', '__getattribute__',
'__gt__', '__hash__', '__init__', '__init_subclass__', '__le__', '__lt__', '__module__', '__ne__', '__new__',
'__reduce__', '__reduce_ex__', '__repr__', '__setattr__', '__sizeof__', '__str__', '__subclasshook__',
'__weakref__', 'get_level', 'num_pokemons', 'set_level']
```

We didn’t define all of these!

Some attributes are defined for all **objects**, and we can override them.

__str__(self)

Returns a String to be printed

print(obj) ⇔ print(obj.__str__()) ⇔ print(str(obj))

- Originally:

```
print(my_pokemon)
>> <pop.Pokemon object at 0x0000019B03804280>
```

- We can implement:

```
class Pokemon:
  # class implementation
  def __str__(self):
    return self.__name
```

Then:

```
print(my_pokemon)
>> Pikachu
```

__repr__(self)

- Returns a String to **represent** the object
- Calling the obj in interpreter ⇔ print(obj.**__repr__()**)
- **str()** is used for creating output for end user while **repr()** is mainly used for debugging and development

__contains__(self, element)

- Checks if an element is in our **object**.
- element **in** **obj** ⇔ **obj.__contains__**(element)

```
class PokemonTrainer:
    # class implementation
    def __contains__(self, pokemon):
        return pokemon in self.pokemons
```

Other Special Operators

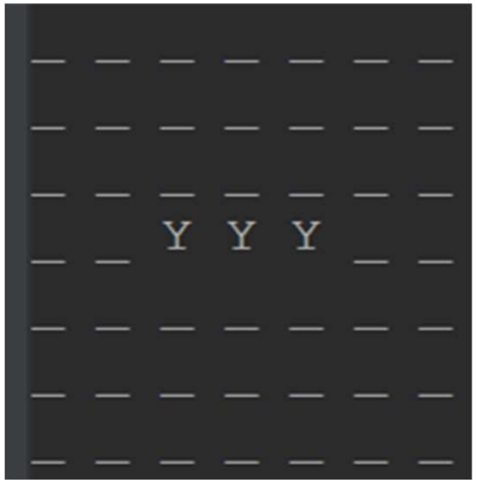
+	object.__add__(self, other)
-	object.__sub__(self, other)
*	object.__mul__(self, other)
//	object.__floordiv__(self, other)
/	object.__div__(self, other)
%	object.__mod__(self, other)
**	object.__pow__(self, other[, modulo])
<<	object.__lshift__(self, other)
>>	object.__rshift__(self, other)
&	object.__and__(self, other)
^	object.__xor__(self, other)
|	object.__or__(self, other)

<	object.__lt__(self, other)
<=	object.__le__(self, other)
==	object.__eq__(self, other)
!=	object.__ne__(self, other)
>=	object.__ge__(self, other)
>	object.__gt__(self, other)

Ex9 – Rush Hour

Ex9 – Rush Hour

- Game objective – get the car out of the crowded parking lot
- The game has different compartments:
  - Board
  - Cars
  - Game
- Each of these has a different responsibility
- The Board “doesn't care” how the car is implemented
  - API



```
_ _ _ _ _ _ _
_ _ _ _ _ _ _
_ _ _ _ _ _ _
_ _ Y Y Y _ _
_ _ _ _ _ _ _
_ _ _ _ _ _ _
_ _ _ _ _ _ _
```