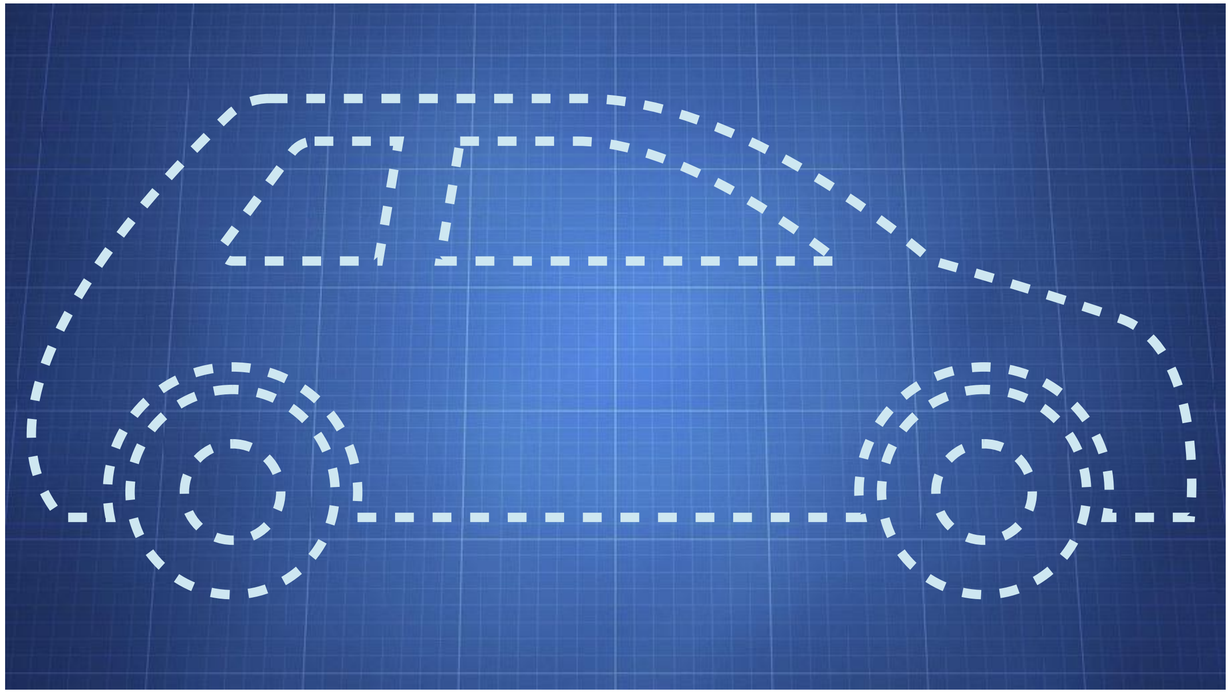
Introduction to Data Science and Programming, Fall 2023

Lecture 16: Object-oriented programming

Instructor: Michael Szell
Oct 27, 2023



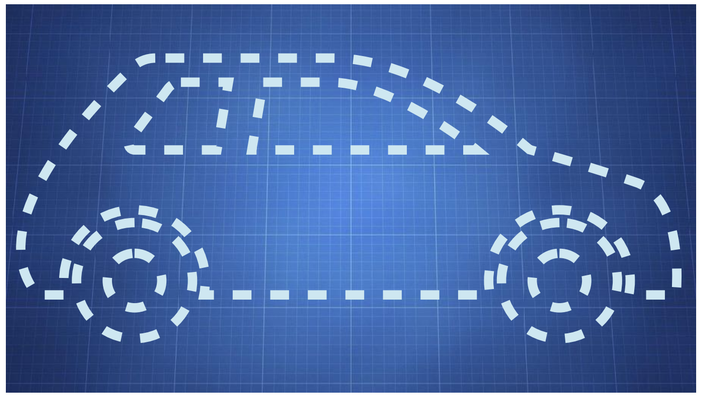
IT UNIVERSITY OF COPENHAGEN

Today you will learn about paradigms, focus on object-oriented

Programming paradigms

Object-oriented design of raquetball simulation

Object-oriented programming



Paradigms classify programming languages based on their features

A series of commands that describe *how* the computations are done.

Paradigms classify programming languages based on their features

A series of commands that describe *how* the computations are done.

```
1 animal = "cat"
2 n = len(animal)
3 animal_reversed = ""
4 for i in range(n-1,-1,-1):
5     animal_reversed += animal[i]
6 animal_reversed
```

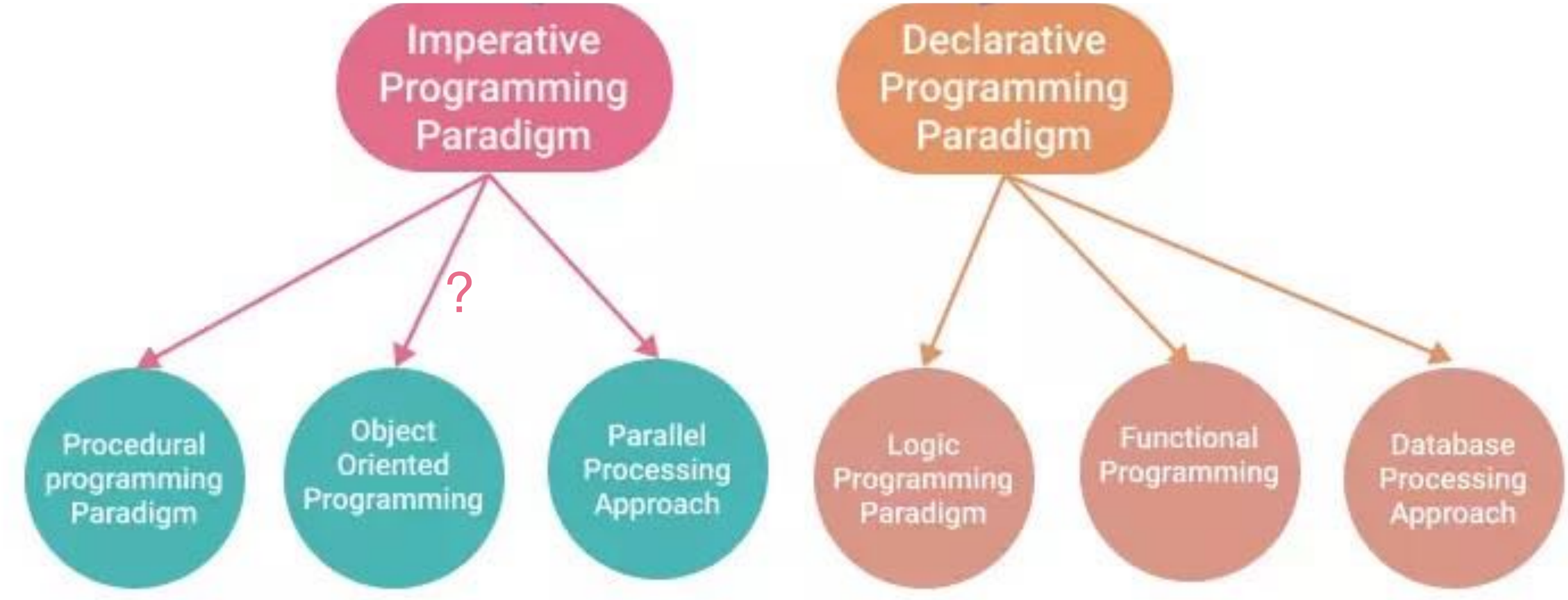
Paradigms classify programming languages based on their features

A series of commands that describe *how* the computations are done.

A series of declarations describing *what* you want.

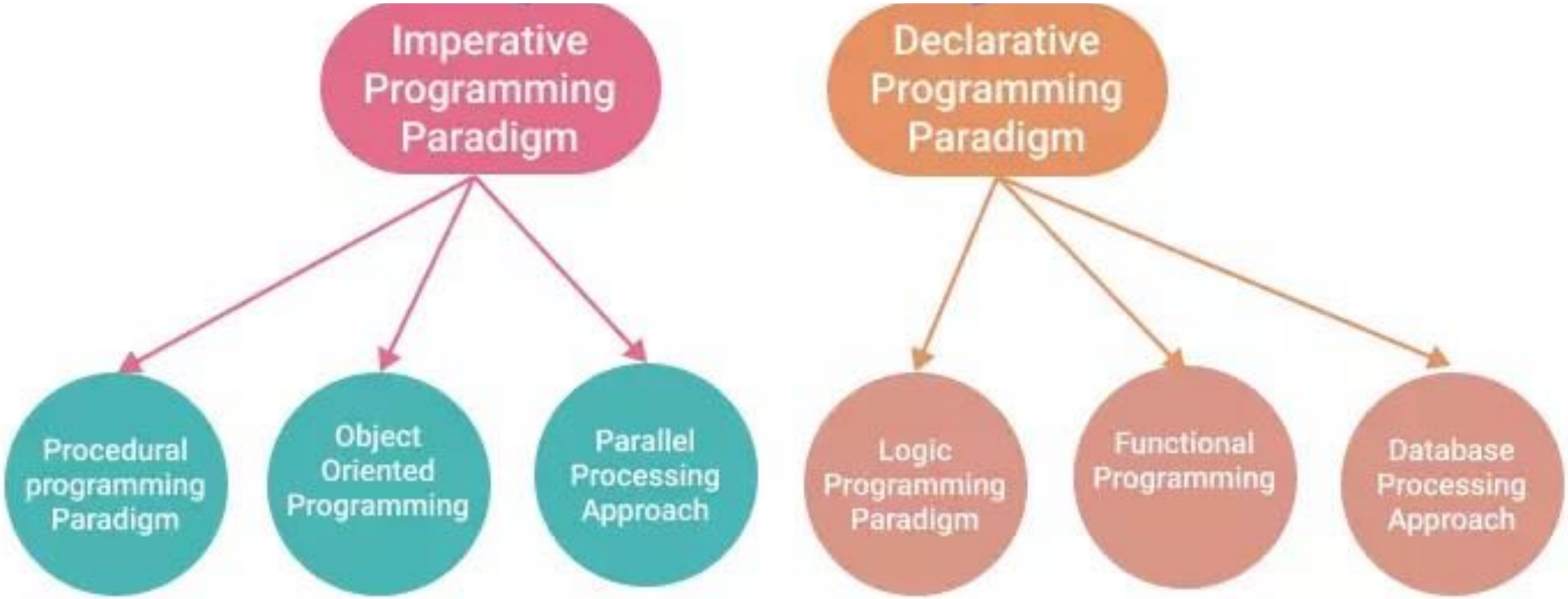
```
1 animal = "cat"
2 n = len(animal)
3 animal_reversed = ""
4 for i in range(n-1,-1,-1):
5     animal_reversed += animal[i]
6 animal_reversed
```

Paradigms classify programming languages based on their features



Functions Objects Functions

Paradigms classify programming languages based on their features



C
BASIC
Pascal
FORTRAN

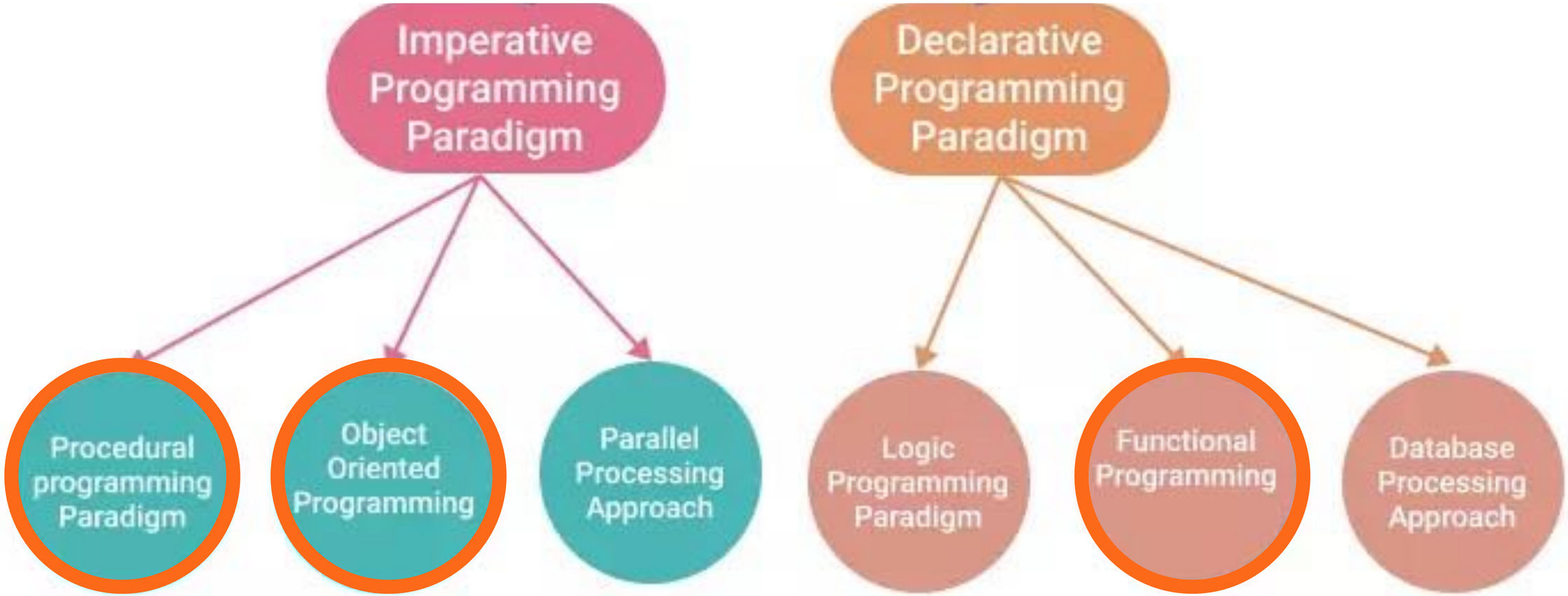
C++
C#
Java
Visual Basic

Prolog

LISP
Haskell

SQL

Paradigms classify programming languages based on their features



C
BASIC
Pascal
FORTRAN
Python

C++
C#
Java
Visual Basic
Python

Prolog

LISP
Haskell
Python

SQL

Python is multi-paradigm

Reversing strings in Python, procedural programming

In [1]:

```
1  def reverse(s):
2      n = len(s)
3      x = ""
4      for i in range(n-1,-1,-1):
5          x += s[i]
6      return x
7
8  animal = "cat"
9  animal_reversed = reverse(animal)
10 animal_reversed
```

Out[1]: 'tac'

Reversing strings in Python, OOP

In [2]:

```
1  from collections import UserString
2
3  class ReversibleString(UserString):
4      def reverse(self):
5          self.data = self.data[::-1]
6
7  animal = ReversibleString("cat")
8  animal.reverse()
9  animal
```

Out[2]: 'tac'

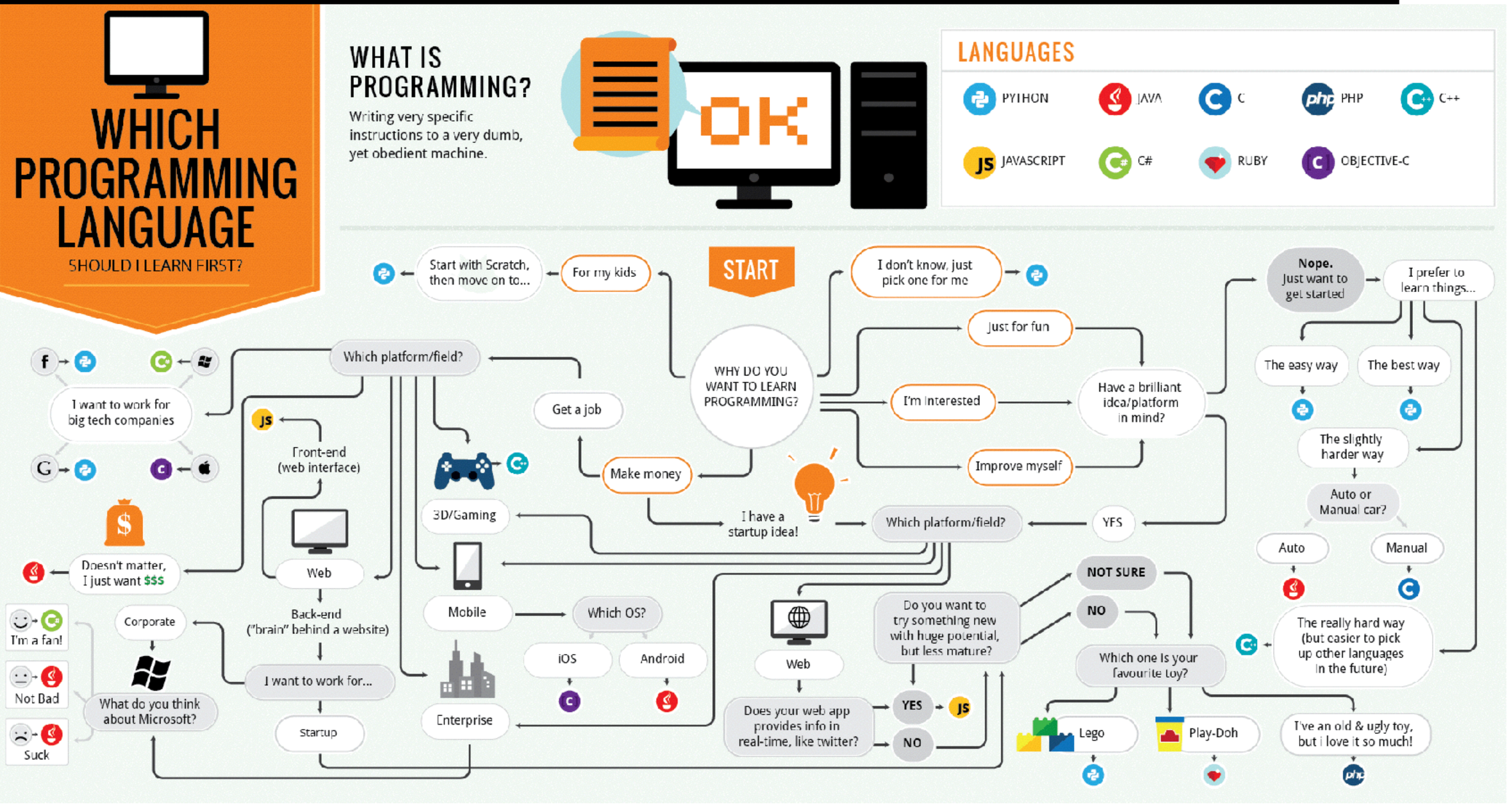
https://realpython.com/reverse-string-python/

Reversing strings in Python, functional programming

```
In [3]:  1  list(map(lambda s: s[::-1], ["cat", "dog", "hedgehog", "gecko"]))

Out[3]: ['tac', 'god', 'gohegdeh', 'okceg']
```

https://realpython.com/python-functional-programming/

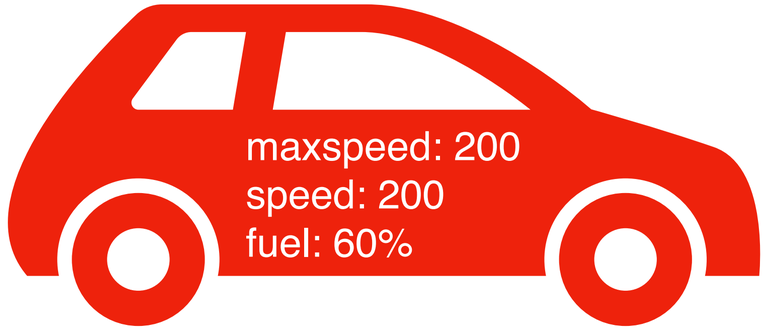
Different paradigms/languages have different use cases

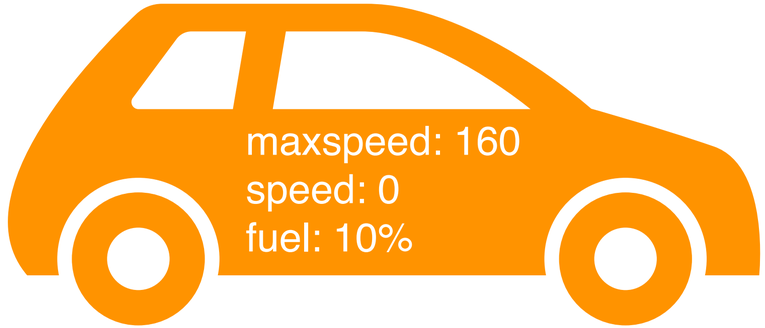


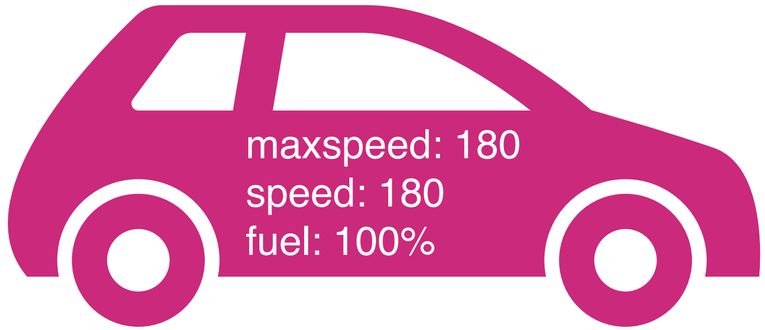
https://www.dailyinfographic.com/flowchart-which-programming-language-should-i-learn-first

Object-oriented programming (OOP)

Objects can contain properties and functionalities



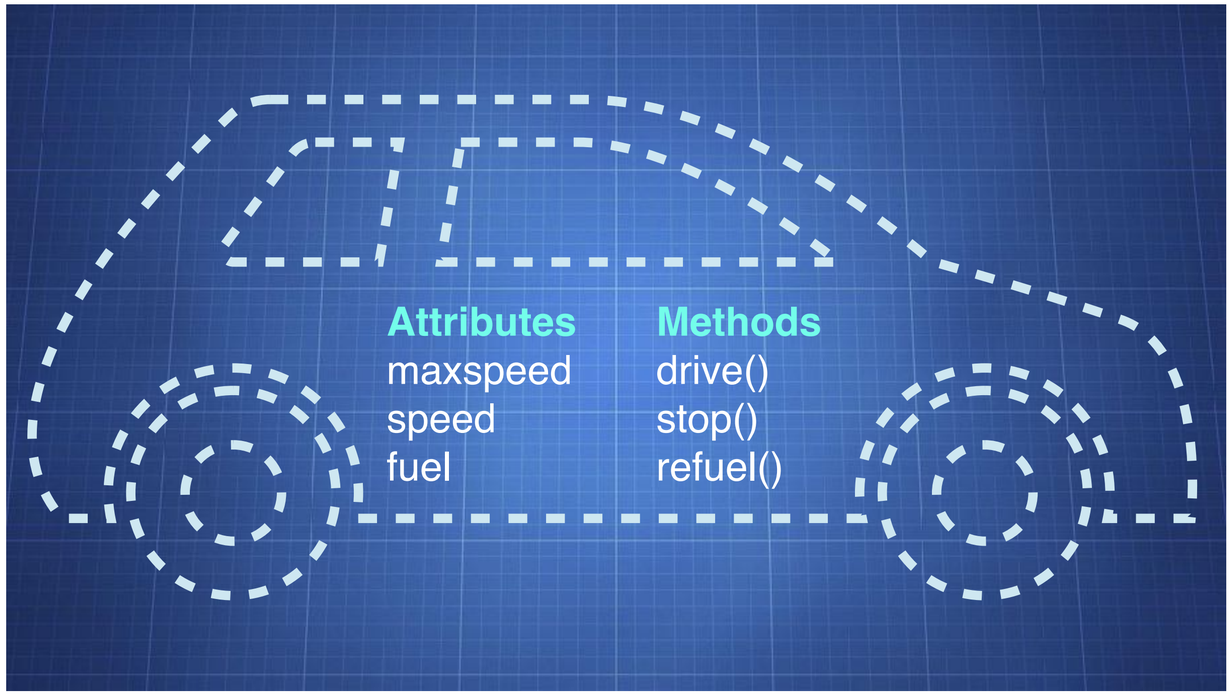




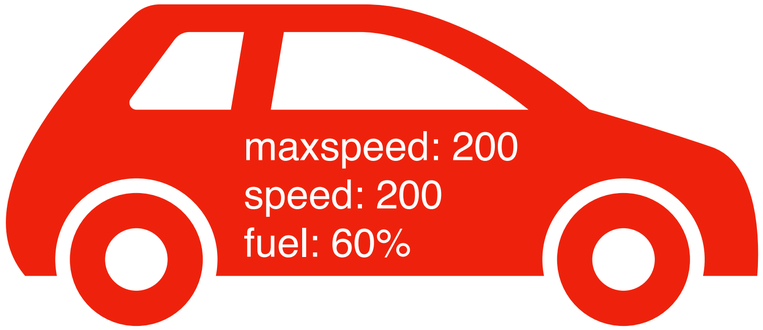
Each car can independently: drive(), stop(), refuel()

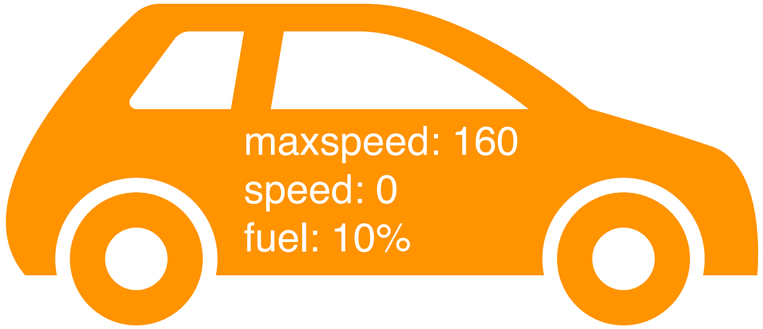
Objects can contain data (attributes) and code (methods)

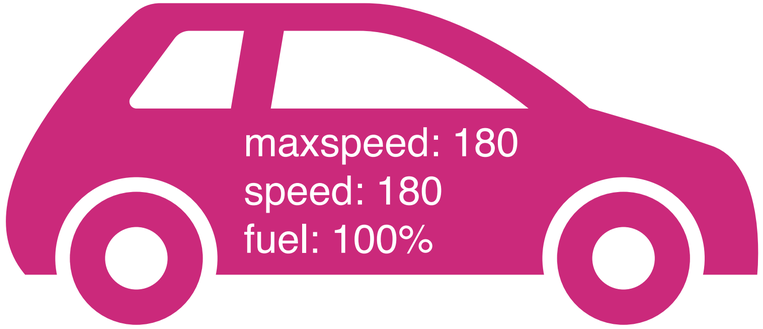
Class



Instances







Each car can independently: drive(), stop(), refuel()

Objects can contain data (attributes) and code (methods)

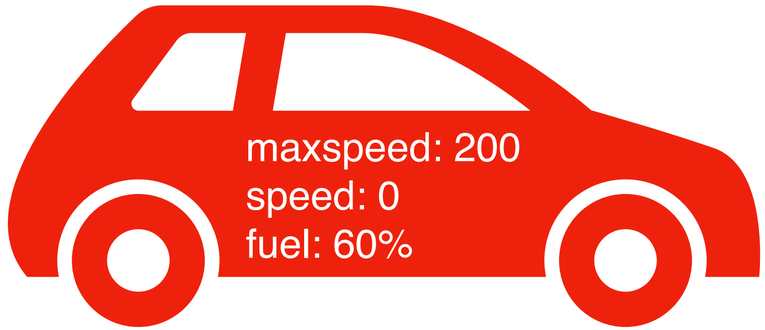
Class

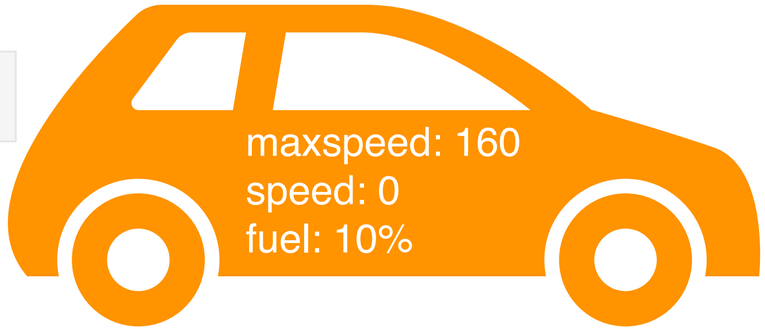
```python
class Car:
    def __init__(self, maxspeed, fuel):
        self.maxspeed = maxspeed
        self.speed = 0
        self.fuel = fuel

    def drive(self):
        self.speed = self.maxspeed
        print("Wroom! Driving at speed " + str(self.speed))

    def stop(self):
        self.speed = 0
        print("Screech!")

    def refuel(self):
        self.fuel = 100
        print("Car has been refueled.")
```

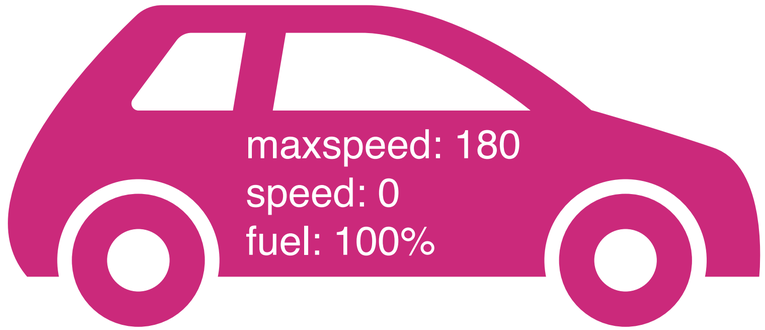
`self` refers to the current instance (`this` in Java)

```python
redcar = Car(200, 60)
orangecar = Car(160, 10)
purplecar = Car(180, 100)
```

Instances







Each car can independently: drive(), stop(), refuel()

The constructor is called when an instance is created

Class

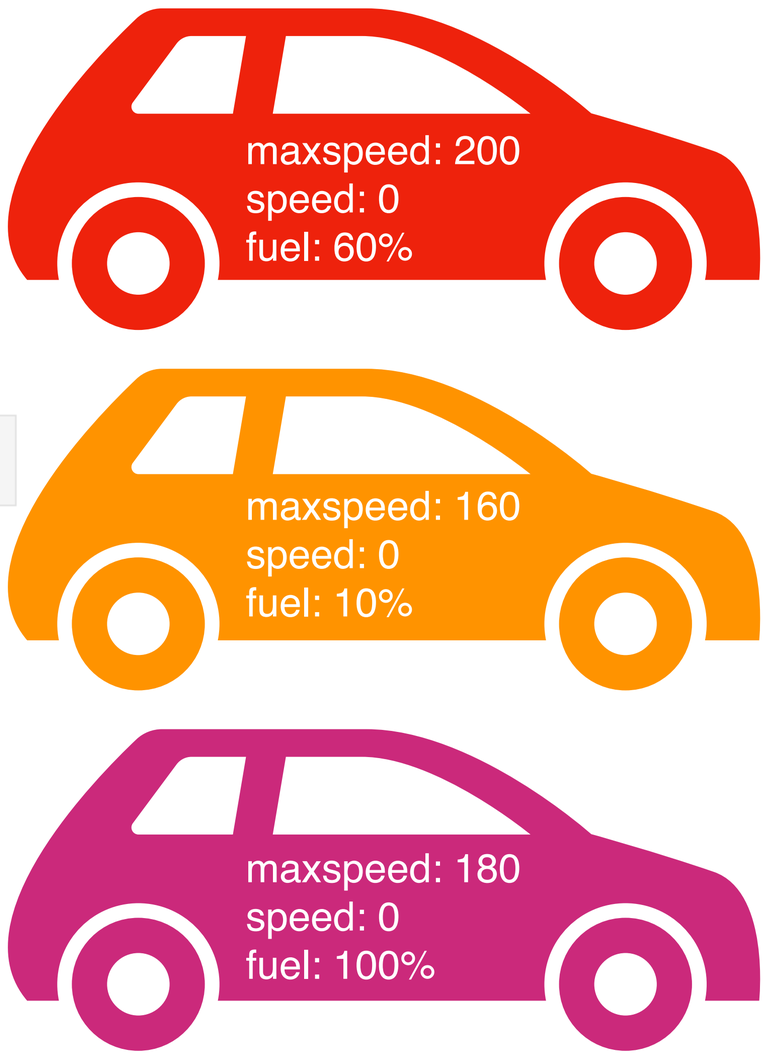
```python
class Car:
    def __init__(self, maxspeed, fuel):
        self.maxspeed = maxspeed
        self.speed = 0
        self.fuel = fuel

    def drive(self):
        self.speed = self.maxspeed
        print("Wroom! Driving at speed " + str(self.speed))

    def stop(self):
        self.speed = 0
        print("Screech!")

    def refuel(self):
        self.fuel = 100
        print("Car has been refueled.")
```

Constructor

`self` refers to the current instance (`this` in Java)

Instances

```python
redcar = Car(200, 60)
orangecar = Car(160, 10)
purplecar = Car(180, 100)
```



Each car can independently: drive(), stop(), refuel()

Class attributes influence all instances

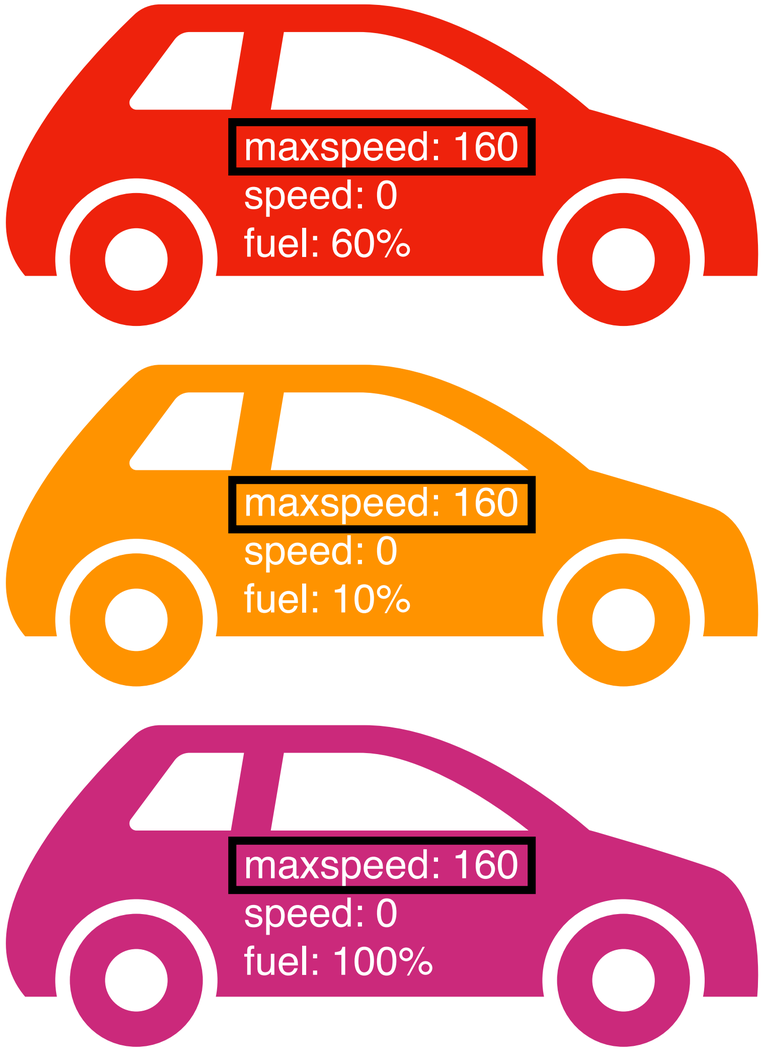
Class

```python
class Car:
    maxspeed = 160

    def __init__(self, fuel):
        self.speed = 0
        self.fuel = fuel

    def drive(self):
        self.speed = Car.maxspeed
        print("Wroom! Driving at speed " + str(self.speed))

    def stop(self):
        self.speed = 0
        print("Screech!")

    def refuel(self):
        self.fuel = 100
        print("Car has been refueled.")
```

`maxspeed` is now a class attribute

```python
redcar = Car(60)
orangecar = Car(10)
purplecar = Car(100)
```

Instances



Each car can independently: drive(), stop(), refuel()

Class attributes influence all instances

Class

```python
class Car:
    maxspeed = 160

    def __init__(self, fuel):
        self.speed = 0
        self.fuel = fuel

    def drive(self):
        self.speed = Car.maxspeed
        print("Wroom! Driving at speed " + str(self.speed))

    def stop(self):
        self.speed = 0
        print("Screech!")

    def refuel(self):
        self.fuel = 100
        print("Car has been refueled.")
```

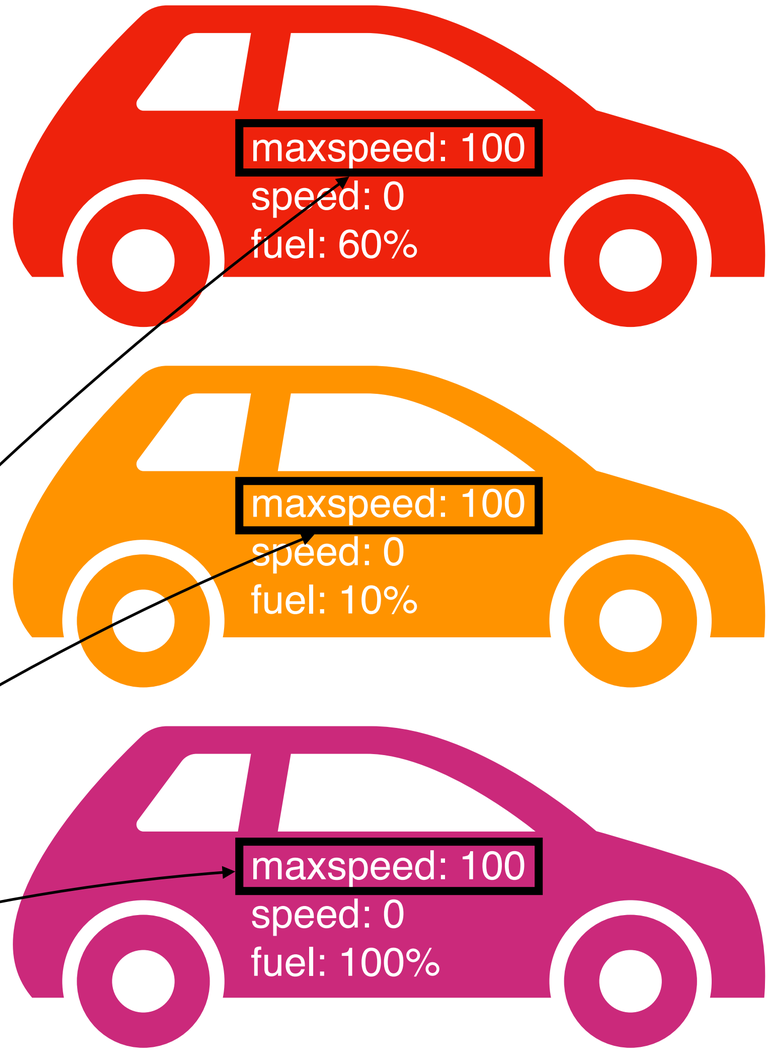
`maxspeed` is now a class attribute

```python
# The EU sets a maximum speed of 100 for all cars
Car.maxspeed = 100
```

Updating it will update it for all instances

```python
redcar = Car(60)
orangecar = Car(10)
purplecar = Car(100)
```

Instances



Each car can independently: drive(), stop(), refuel()

The 3 key concepts of OOP

Encapsulation

Inheritance

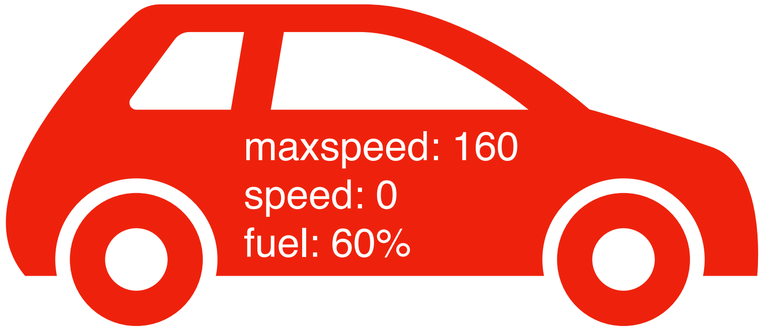
Polymorphism

1) Encapsulation: data and methods are bundled

Class

```python
class Car:
    maxspeed = 160

    def __init__(self, fuel):
        self.speed = 0
        self.fuel = fuel

    def drive(self):
        self.speed = Car.maxspeed
        print("Wroom! Driving at speed " + str(self.speed))

    def stop(self):
        self.speed = 0
        print("Screech!")

    def refuel(self):
        self.fuel = 100
        print("Car has been refueled.")
```

Instances



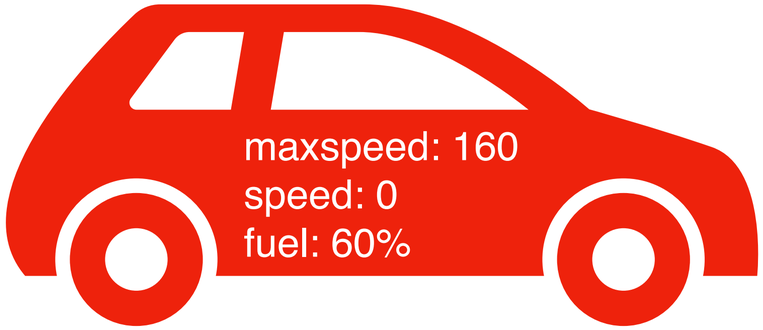
drive(), stop(), refuel()

1) Encapsulation: data and methods are bundled

Class

```
class Car:
    maxspeed = 160

    def __init__(self, fuel):
        self.speed = 0
        self.fuel = fuel

    def drive(self):
        self.speed = Car.maxspeed
        print("Wroom! Driving at speed " + str(self.speed))

    def stop(self):
        self.speed = 0
        print("Screech!")

    def refuel(self):
        self.fuel = 100
        print("Car has been refueled.")
```

Instances



drive(), stop(), refuel()

```
redcar.speed = 200
```

Don't tamper with an object's data directly!

You must use its methods so it behaves as intended

Abstraction

2) Inheritance: Subclasses can inherit

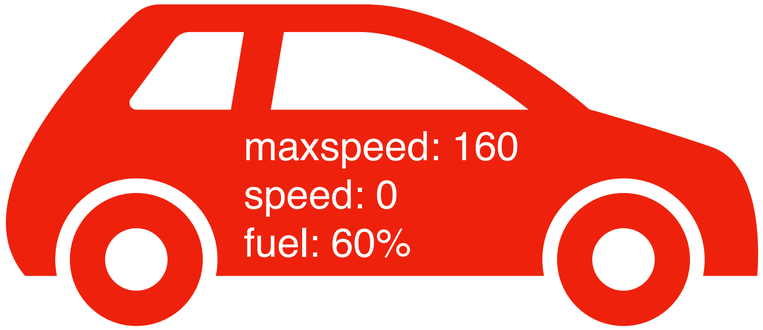
Parent Class

```
class Car:
    maxspeed = 160

    def __init__(self, fuel):
        self.speed = 0
        self.fuel = fuel

    def drive(self):
        self.speed = Car.maxspeed
        print("Wroom! Driving at speed " + str(self.speed))

    def stop(self):
        self.speed = 0
        print("Screech!")

    def refuel(self):
        self.fuel = 100
        print("Car has been refueled.")
```

Child Class

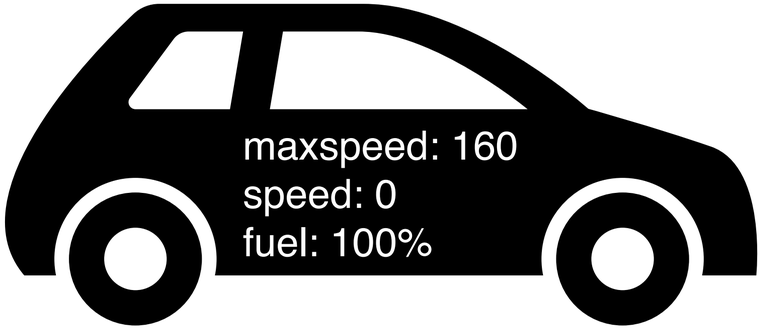
```
class Supercar(Car):
    turbospeed = 300

    def turboboost(self):
        self.speed = Supercar.turbospeed
        print("Wroooooom! Driving at speed " + str(self.speed))
```

Instances



drive(), stop(), refuel()

```
mysupercar = Supercar(100)
```



drive(), stop(), refuel(), turboboost()

3) Polymorphism: Methods work on different data types

```
def plus(a, b):
    return a + b
```

```
print(plus(int(3), float(3.4)))
print(plus([1,2,3], [4,5]))
print(plus("abra", "kadabra"))
```

```
6.4
[1, 2, 3, 4, 5]
abrakadabra
```

3) Polymorphism: Methods work on different data types

```
def plus(a, b):
    return a + b

def plus_integers(a,b):
    """This code just works when a and b are integers
    """

def plus_floats(a,b):
    """This code just works when a and b are floats
    """
    ...

print(plus(int(3), float(3.4)))
print(plus([1,2,3], [4,5]))
print(plus("abra", "kadabra"))
```

```
6.4
[1, 2, 3, 4, 5]
abrakadabra
```

The 3 key concepts of OOP

Encapsulation

Data and its methods are bundled. You don't want to tamper with an object's data directly.

Inheritance

A child class can inherit attributes and methods from a parent class.

Polymorphism

Methods work on different data types.

`self` refers to the current instance (`this` in Java)

Jupyter

Do we need OOP in Data Science?

- Everything is an object in Python
- Easier code re-use, debugging, testing
- Can facilitate parallel development in teams
- Great for simulations/modeling (agent-based)

Do we need OOP in Data Science?

- Everything is an object in Python
- Easier code re-use, debugging, testing
- Can facilitate parallel development in teams
- Great for simulations/modeling (agent-based)

- Code base could become too large / bloated
- Harder to learn
- Can increase runtime or code duplication

Take-home message 1

Programming paradigms are different ways of thinking about a problem and of implementing a solution

In Python everything is an object, but it is often used just procedurally

Take-home message 2: The 3 key concepts of OOP

Encapsulation

Data and its methods are bundled. You don't want to tamper with an object's data directly.

Inheritance

A child class can inherit attributes and methods from a parent class.

Polymorphism

Methods work on different data types.

`self` refers to the current instance (`this` in Java)