Programming For Data Science

Advanced Topics - OOP:

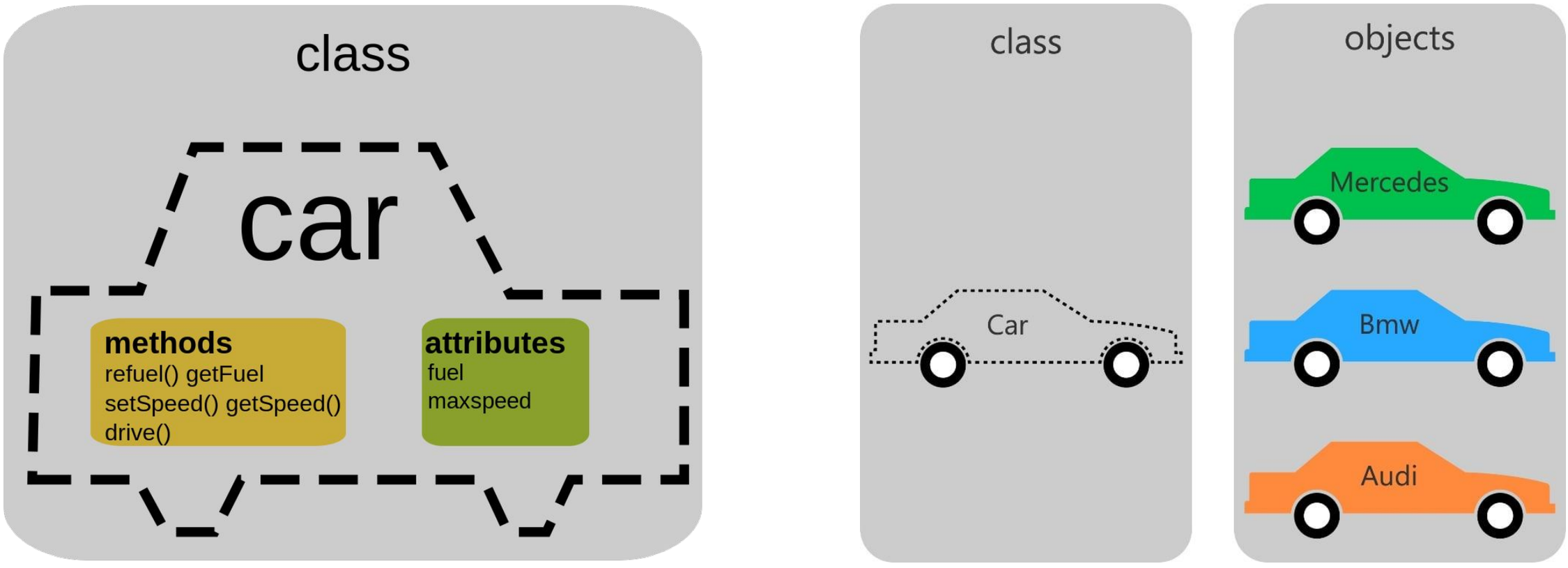
Classes & Objects

Giulio Rossetti
giulio.rossetti@isti.cnr.it

Credits: Giuseppe Prencipe
giuseppe.prencipe@unipi.it

Classes and objects

In Python everything is an object
….everything is an instance of a class….



A class is a **blueprint**, a *model* for its objects.

….used to define properties (*attributes*) and behaviour (*methods*)

Vehicle example

- car is an object (or *instance*) of the class Vehicle

```
car = Vehicle()
print (car) #<__main__.Vehicle instance at 0x7fb1de6c2638>
```

Object creation

Vehicle example

- Vehicle class has 4 attributes:
  - n. of wheels, type of tank, seating capacity, max speed

Class names typically start with capital letter

```
class Vehicle:
```

Class definition

```
    def __init__(self, number_of_wheels, type_of_tank, seating_capacity, maximum_velocity):
        self.number_of_wheels = number_of_wheels
        self.type_of_tank = type_of_tank
        self.seating_capacity = seating_capacity
        self.maximum_velocity = maximum_velocity
```

Initialisation method

(it always exists, even if not defined explicitly — **aka constructor**

Vehicle example

- Vehicle class has 4 attributes:
  - n. of wheels, type of tank, seating capacity, max speed

self: reference to the object itself

(always first parameter of every method you create)

```
class Vehicle:

    def __init__(self, number_of_wheels, type_of_tank, seating_capacity, maximum_velocity):
        self.number_of_wheels = number_of_wheels
        self.type_of_tank = type_of_tank
        self.seating_capacity = seating_capacity
        self.maximum_velocity = maximum_velocity
```

Vehicle example

- Vehicle class has 4 attributes:
  - n. of wheels, type of tank, seating capacity, max speed

```
class Vehicle:

    def __init__(self, number_of_wheels, type_of_tank, seating_capacity, maximum_velocity):
        self.number_of_wheels = number_of_wheels
        self.type_of_tank = type_of_tank
        self.seating_capacity = seating_capacity
        self.maximum_velocity = maximum_velocity
```

Definition of class attributes

Vehicle example

- Vehicle class has 4 attributes:
  - n. of wheels, type of tank, seating capacity, max speed

```
class Vehicle:


    def __init__(self, number_of_wheels, type_of_tank, seating_capacity, maximum_velocity):
        self.number_of_wheels = number_of_wheels
        self.type_of_tank = type_of_tank
        self.seating_capacity = seating_capacity
        self.maximum_velocity = maximum_velocity


car = Vehicle(4, 'electric', 5, 250)
```

Object creation

Vehicle example

- How can we access the attributes of the car object?
    - By *sending a message to the object*, through a method (object's behaviour)

```
class Vehicle:

    def __init__(self, number_of_wheels, type_of_tank, seating_capacity, maximum_velocity):

        #omitted


def get_number_of_wheels(self):

        return self.number_of_wheels


    def set_number_of_wheels(self, number):

        self.number_of_wheels = number
```

Getter

Setter

Vehicle example

- How can we access the attributes of the car object?
  - By *sending a message to the object*, through a method (object's behaviour)

```
class Vehicle:
    def __init__(self, number_of_wheels, type_of_tank, seating_capacity, maximum_velocity):
        #omitted
    def get_number_of_wheels(self):
        return self.number_of_wheels
    def set_number_of_wheels(self, number):
        self.number_of_wheels = number


tesla_model_s = Vehicle(4, 'electric', 5, 250)
print(tesla_model_s.get_number_of_wheels()) # 4
tesla_model_s.set_number_of_wheels(3) # setting number of wheels to 3
```

You can access directly attributes,
to read and write them

```
tesla_model_s.number_of_wheels = 8
```

Vehicle example

- How to display the content of a class?
  - print() produces ugly results!

```
class Vehicle:
    def __init__(self, number_of_wheels, type_of_tank, seating_capacity, maximum_velocity):
        #omitted


tesla_model_s = Vehicle(4, 'electric', 5, 250)
print(tesla_model_s) #<__main__.Vehicle instance at 0x7fb1de6c2638>
```

Vehicle example

- How to display the content of a class?
  - print() produces ugly results!

```
class Vehicle:
    def __init__(self, number_of_wheels, type_of_tank, seating_capacity, maximum_velocity):
        #omitted

    def __repr__(self):
        return "({}, {}, {}, {})".format(self.number_of_wheels, self.type_of_tank, self_seating_capacity, self_maximum_velocity)

tesla_model_s = Vehicle(4, 'electric', 5, 250)
print(tesla_model_s) #(4, electric, 5, 250)
```

Usually you display all elements that allow to fully reconstruct the object

Vehicle example

- There exists also a __str__() method, typically used to display just a part of the info
  - If __str__() is not defined, it is the same as __repr__()

```
class Vehicle:
    def __init__(self, number_of_wheels, type_of_tank, seating_capacity, maximum_velocity):
        #omitted
    def __repr__(self):
        return "({}, {}, {}, {})".format(self.number_of_wheels, self.type_of_tank, self_seating_capacity, self_maximum_velocity)


tesla_model_s = Vehicle(4, 'electric', 5, 250)
print(tesla_model_s) #(4, electric, 5, 250)
```

Usually you display all elements that allow to fully reconstruct the object — always do it!

Vehicle example

- Beside __init__, __repr__, __str__, setters and getters, you can define your own methods

```
class Vehicle:

    def __init__(self, number_of_wheels, type_of_tank, seating_capacity, maximum_velocity):

        #omitted


    def make_noise(self):

        print("VRUUUUUM")



tesla_model_s = Vehicle(4, 'electric', 5, 250)

tesla_model_s.make_noise()
```

Vehicle example

- Objects can be part of other objects!

```
class Engine:
    def __init__(self, type):
        self.type = type


class Vehicle:
    def __init__(self, number_of_wheels, type_of_tank, seating_capacity, maximum_velocity):
        #omitted


electric_engine = Engine('electric')
tesla_model_s = Vehicle(4, electric_engine, 5, 250)
print(tesla_model_s)
```

Assuming we have __repr__ for Vehicle, What do you get here?

Vehicle example

- Objects can be part of other objects!

```
class Engine:
    def __init__(self, type):
        self.type = type


class Vehicle:
    def __init__(self, number_of_wheels, type_of_tank, seating_capacity, maximum_velocity):
        #omitted
electric_engine = Engine('electric')
tesla_model_s = Vehicle(4, electric_engine, 5, 250)
print(tesla_model_s) #(4, <__main__.Engine object at 0x119985a90>, 5, 250)
```

Vehicle example

- Objects can be part of other objects!
  - What happens when an object that is an attribute is modified?

```
class Engine:
    def __init__(self, type):
        self.type = type


class Vehicle:
    def __init__(self, number_of_wheels, type_of_tank, seating_capacity, maximum_velocity):
        #omitted
electric_engine = Engine('electric')
tesla_model_s = Vehicle(4, electric_engine, 5, 250)
print(tesla_model_s)
electric_engine.type = ‘gas'
print(tesla_model_s)
```

Assuming we have __repr__ for Engine and Vehicle, what do you expect here?

Vehicle example

- Objects can be part of other objects!
  - What happens when an object that is an attribute is modified?

```
class Engine:
    def __init__(self, type):
        self.type = type


class Vehicle:
    def __init__(self, number_of_wheels, type_of_tank, seating_capacity, maximum_velocity):
        #omitted
electric_engine = Engine('electric')
tesla_model_s = Vehicle(4, electric_engine, 5, 250)
print(tesla_model_s) #(4, electric, 5, 250)
electric_engine.type = ‘gas'
print(tesla_model_s) #(4, gas, 5, 250)
```

Objects are passed by reference!

(no copy is done)

Vehicle example

- Objects can be part of other objects!
  - What do to if we want a copy?

```
class Engine:
    def __init__(self, type):
        self.type = type


class Vehicle:
    def __init__(self, number_of_wheels, type_of_tank, engine, seating_capacity, maximum_velocity):
        #omitted
electric_engine = Engine('electric')
tesla_model_s = Vehicle(4, electric_engine, 5, 250)
print(tesla_model_s) #(4, electric, 5, 250)
electric_engine.type = ‘gas'
print(tesla_model_s) #(4, gas, 5, 250)
```

How to change to avoid pass by reference?

Vehicle example

- Objects can be part of other objects!
  - What do to if we want a copy?

```
class Engine:
    def __init__(self, type):
        self.type = type


class Vehicle:
    def __init__(self, number_of_wheels, type_of_tank, seating_capacity, maximum_velocity):
        #omitted
        self.type_of_tank = copy(type_of_tank)
electric_engine = Engine('electric')
tesla_model_s = Vehicle(4, electric_engine, 5, 250)
print(tesla_model_s) #(4, electric, 5, 250)
electric_engine.type = ‘gas'
print(tesla_model_s) #(4, electric, 5, 250)
```

Shallow copy (need to import copy)

— for deep copy, use deepcopy()

(Proposed) Exercises

- Modify the constructor of last class `Vehicle` (that refers an `Engine` object) so that it initialises the number of wheels, seating capacity, and max speed attributes when some of them are not passed as argument when the object is created
- Implement a class `Point` that represents a coordinate on a 2D plane
  - Implement the method `translate()` that translates it by `dx` and `dy`
  - Implement the methods `distance_from_origin()`
  - Implement the method `distance()` which takes another point as parameter and computes the distance between the two points
- Implement a class `Rectangle`, that has as attributes a `Point` (representing the top-left corner), a width and a height
  - Implement a method which computes the area of a rectangle
- Implement a class `Circle` constructed by a radius and two methods which will compute the area and the perimeter of a circle
- **All** exercises in Chapter 20 of the reference book

Programming For Data Science

Advanced Topics - OOP:

Overloading

Giulio Rossetti
giulio.rossetti@isti.cnr.it

Credits: Giuseppe Prencipe
giuseppe.prencipe@unipi.it

Operator overloading

- We can define functions so that Python's built-in operators can be used with our class

Operator	Method
+	`__add__(self, other)`
-	`__sub__(self, other)`
*	`__mul__(self, other)`
/	`__truediv__(self, other)`

Operator	Method
==	`__eq__(self, other)`
!=	`__ne__(self, other)`
<	`__lt__(self, other)`
>	`__gt__(self, other)`

```
class Point:
    def __init__(self, x = 0, y = 0):
        self.x = x
        self.y = y
    def __add__(self, other):
        return Point(self.x+other.x, self.y+other.y)
    def __str__(self):
        return "(" + str(self.x) + ", " + str(self.y) + ")"

# main
p1 = Point(4, 2)
p2 = Point(3, 1)
print (p1+p2)
```

Operator overloading

- Unary operators (work only on the object)
  - __neg__()
  - __abs__()
  - ….
- Sequences: You can implement sequences classes (similar to tuples, lists, etc.)
    - __len__()
    - __getitem__()
    - __contains__()
    - ….

(Proposed) Exercises

- Overload the operators -, *, /, ==, and < for the class Point
- Add to the Rectangle class operators to test for equality of rectangles (two rectangles are equal if they have exactly the same shape), and greater/smaller operators (a rectangle is smaller than another rectangle if it has a smaller surface area). Test the new operators
- **All** exercises in Chapter 21 of the reference book

Programming For Data Science

Advanced Topics - OOP:

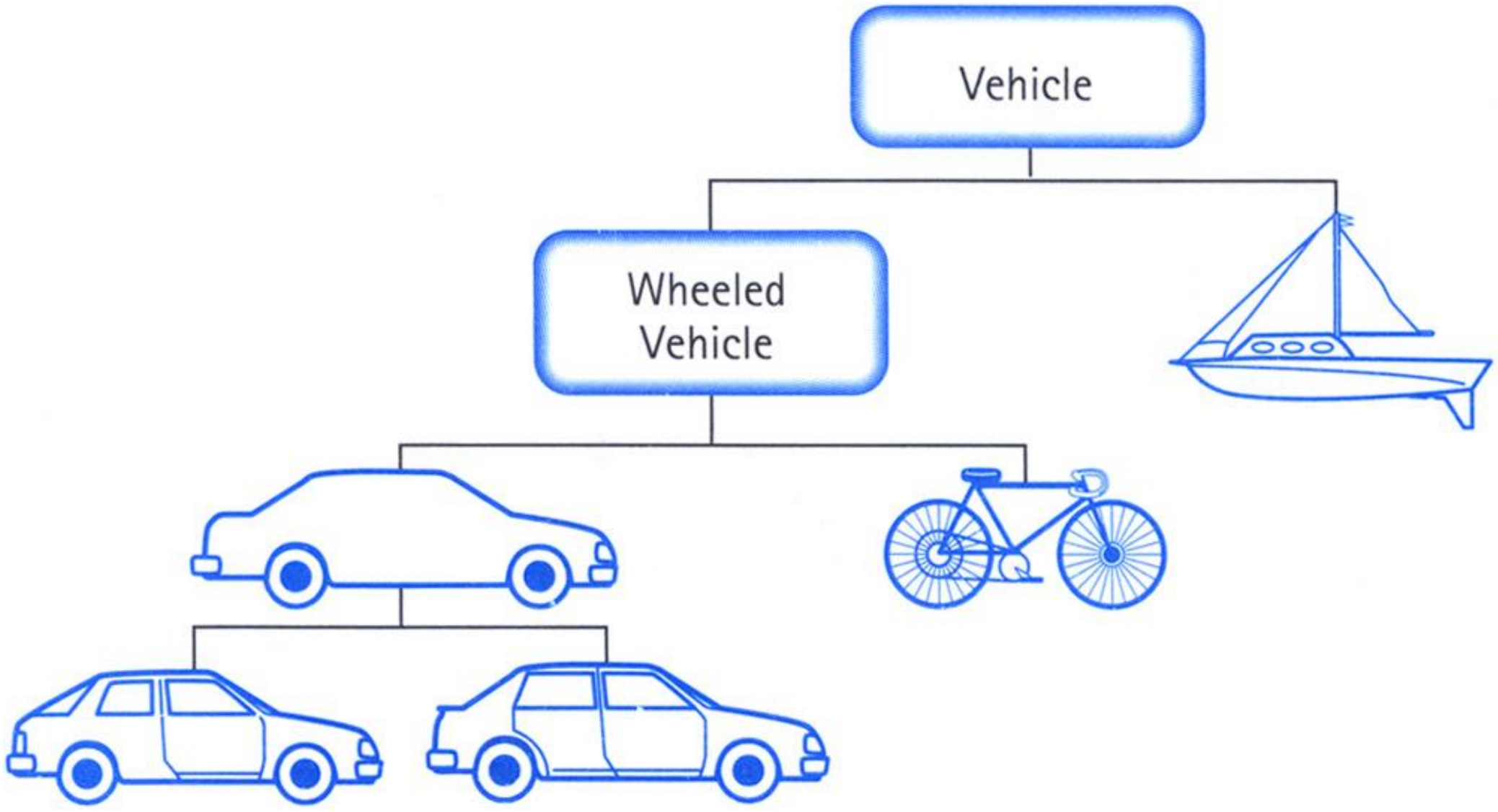
Inheritance

Giulio Rossetti
giulio.rossetti@isti.cnr.it

Credits: Giuseppe Prencipe
giuseppe.prencipe@unipi.it

Inheritance

- Create classes from existing ones



Person example

- Person is a super-class
  - Employee is also a person

```
class Person:
    def __init__(self, first, last):
        self.firstname = first
        self.lastname = last

    def name(self):
        return self.firstname + " " + self.lastname

class Employee(Person):
    pass
```

super-class

null operation

Person example

- Person is a super-class
  - Employee is a person, with an extra feature

```
class Person:
    def __init__(self, first, last):
        self.firstname = first
        self.lastname = last

    def name(self):
        return self.firstname + " " + self.lastname

class Employee(Person):
    def __init__(self, first, last, staffnum):
        Person.__init__(self,first, last)
        self.staffnumber = staffnum

    def GetEmployee(self):
        return self.name() + ", " +  self.staffnumber
```

super-class

overrides the super-class method

method specific for employee

Person example

- Person is a super-class
  - Employee is a person, with an extra feature

```
class Person:
    def __init__(self, first, last):
        self.firstname = first
        self.lastname = last

    def name(self):
        return self.firstname + " " + self.lastname

class Employee(Person):
    def __init__(self, first, last, staffnum):
        super().__init__(self,first, last)
        self.staffnumber = staffnum

    def GetEmployee(self):
        return self.name() + ", " +  self.staffnumber
```

super() refers to the super-class

name() defined in the super-class

Person example

- Person is a super-class
  - Employee is a person, with an extra feature

```
class Person:
    def __init__(self, first, last):
        self.firstname = first
        self.lastname = last

    def name(self):
        return self.firstname + " " + self.lastname

class Employee(Person):
    def __init__(self, first, last, staffnum):
        super().__init__(self,first, last)
        self.staffnumber = staffnum

    def GetEmployee(self):
        return self.Name() + ", " +  self.staffnumber

x = Person("Marge", "Simpson")
y = Employee("Homer", "Simpson", "1007")
print(x.name()) #Marge Simpson
print(y.GetEmployee()) #Homer Simpson, 1007
```

Person example

- Person is a super-class
  - Employee is a person, with an extra feature

```
class Person:
    def __init__(self, first, last):
        self.firstname = first
        self.lastname = last

    def __str__(self):
        return self.firstname + " " + self.lastname

class Employee(Person):
    def __init__(self, first, last, staffnum):
        super().__init__(self,first, last)
        self.staffnumber = staffnum

x = Person("Marge", "Simpson")
y = Employee("Homer", "Simpson", "1007")
print(x) #Marge Simpson
print(y) #Homer Simpson
```

inheritance!

Person example

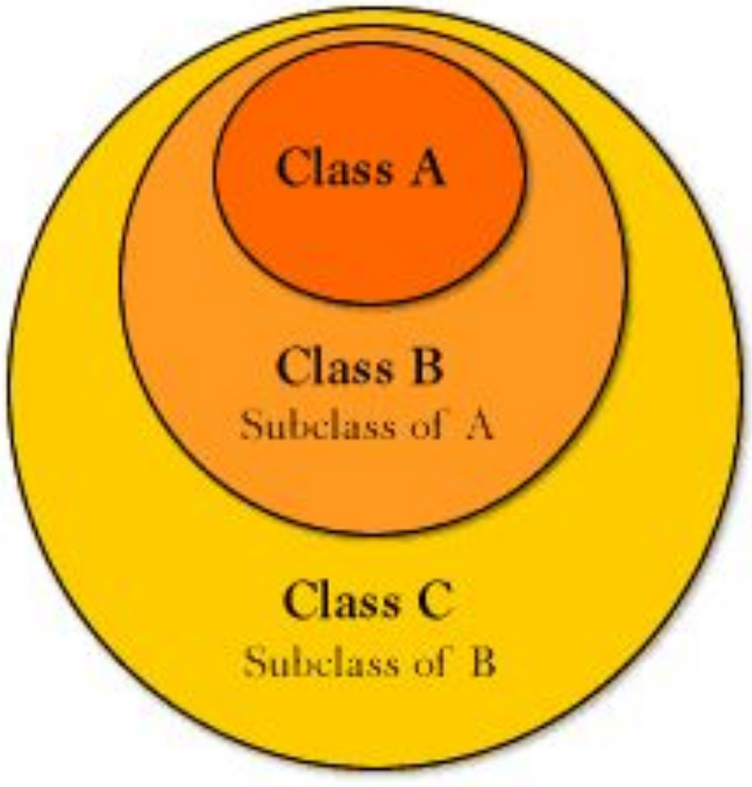
- Person is a super-class
  - Employee is a person, with an extra feature

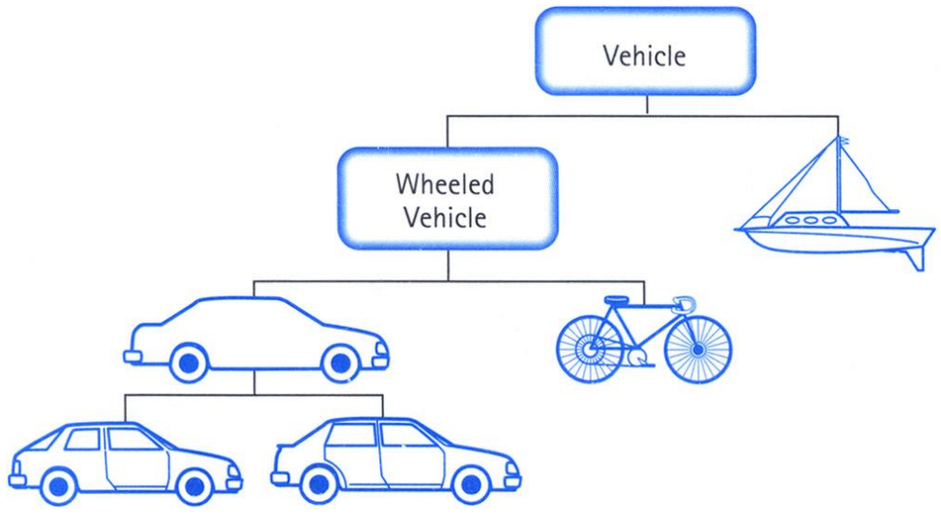
```
class Person:
    def __init__(self, first, last):
        self.firstname = first
        self.lastname = last
   def __str__(self):
        return self.firstname + " " + self.lastname

class Employee(Person):
    def __init__(self, first, last, staffnum):
        super().__init__(self,first, last)
        self.staffnumber = staffnum
def __str__(self):
        return super().__str__() + ", " +  self.staffnumber
x = Person("Marge", "Simpson")
y = Employee("Homer", "Simpson", "1007")
print(x) #Marge Simpson
print(y) #Homer Simpson, 1007
```

override!

Inheritance

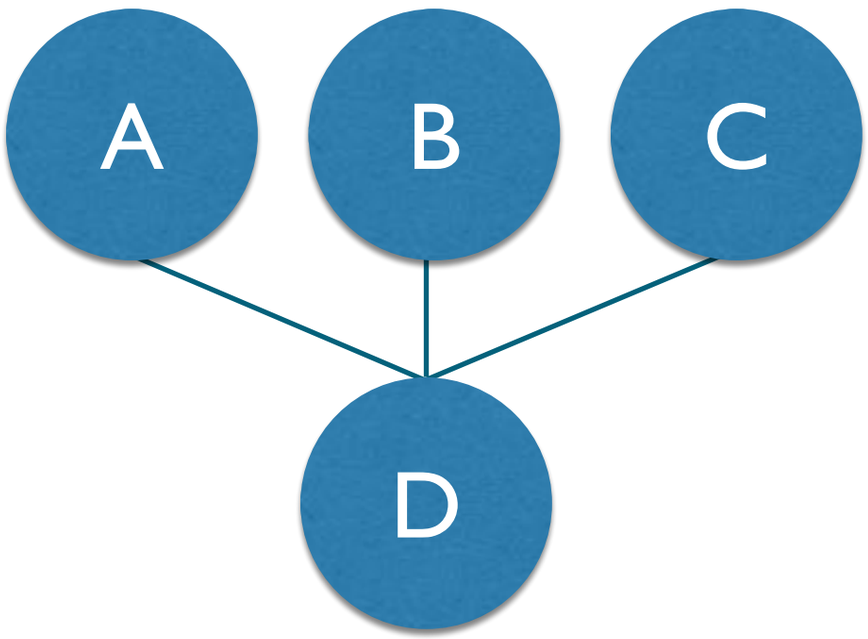
- Sub-class is a specialisation of the super-class
  - You get all the features of the super-class
  - You can extend (i.e., add) or override (i.e., change the meaning as defined in the super-class) them





Multiple inheritance

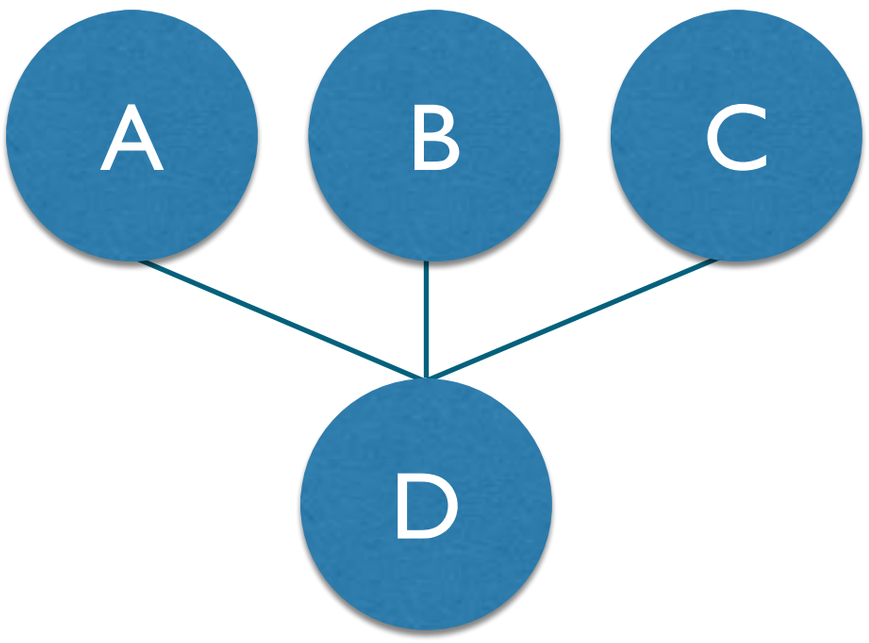
- Classes can inherit from multiple classes
    - When a method is called, to decide which method implementation to use, Python first checks whether it exists in the class for which the method is called itself. If it is not there, it checks all the superclasses, from left to right




```
class D(A, B, C):
        #….
```

Multiple inheritance

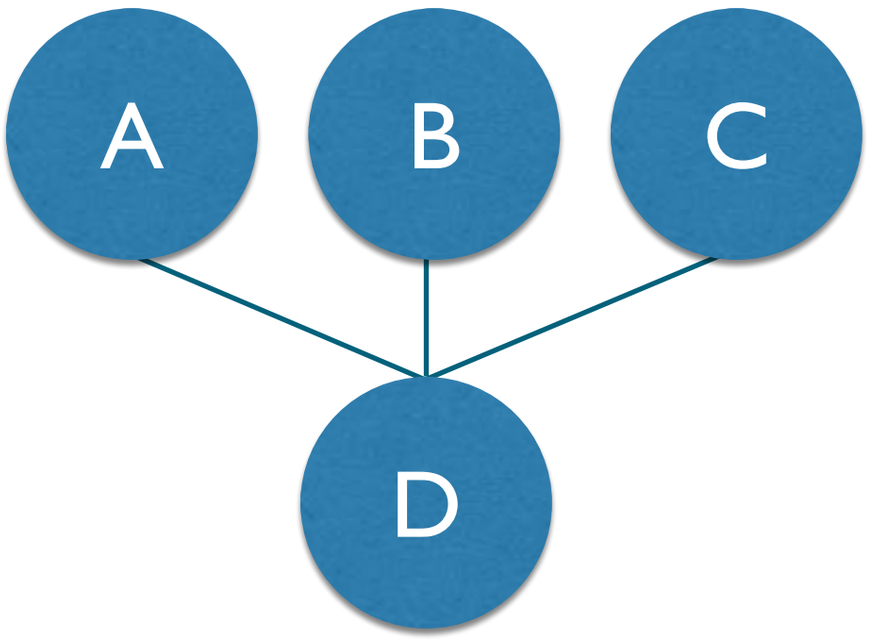
- Classes can inherit from multiple classes
    - If you want to call a method from a superclass,
      you have to tell Python which superclass you wish to call




```
class D(A, B, C):
    def __init__(self):
        #A.__init__(self)
        #B.__init__(self)
        #C.__init__(self)
```

Multiple inheritance

- Classes can inherit from multiple classes
  - If you want to call a method from a superclass,
    you have to tell Python which superclass you wish to call
  - You can do that also using `super()` — tricky!




```
class D(A, B, C):
    def __init__(self):
        #super().__init__()
```

Multiple inheritance

```
class A:
    def __init__(self):
        print("A")

class B:
    def __init__(self):
        print("B")

class C:
    def __init__(self):
        print("C")

class D(A,B,C):
    def __init__(self):
        super(D, self).__init__()

a = D()
```

super takes 2 params:

the type of the class and

the reference to the object

Multiple inheritance

```
class A:
    def __init__(self):
        print("A")

class B:
    def __init__(self):
        print("B")

class C:
    def __init__(self):
        print("C")

class D(A,B,C):
    def __init__(self):
        super(D, self).__init__()

a = D()
```

What do we get printed here?

Multiple inheritance

```
class A:
    def __init__(self):
        print("A")

class B:
    def __init__(self):
        print("B")

class C:
    def __init__(self):
        print("C")

class D(A,B,C):
    def __init__(self):
        super(D, self).__init__()

a = D()
```

What do we get printed here?

A (is the first super class that is checked)

Multiple inheritance

```
class A:
    def __init__(self):
        print("A")

class B:
    def __init__(self):
        print("B")

class C:
    def __init__(self):
        print("C")

class D(B,A,C):
    def __init__(self):
        super(D, self).__init__()

a = D()
```

And now?

Multiple inheritance

```
class A:
    def __init__(self):
        print("A")

class B:
    def __init__(self):
        print("B")

class C:
    def __init__(self):
        print("C")

class D(B,A,C):
    def __init__(self):
        super(D, self).__init__()

a = D()
```

And now?

B

Multiple inheritance always tricky….better to avoid it, unless really necessary!

Interfaces

- An **interface** is a class that specifies attributes and methods without an actual implementation of the methods

```
class Vehicle:
    def __init__(self):
        self.startpoint = []
        self.endpoints = []
        self.verb = “”
        self.name = “”
    def __str__(self):
        return self.name
    def isStartPoint(self, p):
        return NotImplemented
    def isEndPoint(self, p):
        return NotImplemented
    def travelVerb(self):
        return NotImplemented
```

No method implemented

They will be defined

when sub-classes created

(Suggested) Exercises

- **All** exercises in Chapter 22 of the reference book

Programming For Data Science

Advanced Topics:

Iterators & Generators

Giulio Rossetti
giulio.rossetti@isti.cnr.it

Credits: Giuseppe Prencipe
giuseppe.prencipe@unipi.it

Iterators

- To use your classes in for….in…. statements
  - List, strings, and dictionaries are all “iterables”, and can be used in for expressions

Iterators

- To use your classes in for….in…. statements
  - An *iterator* (created with iter()) is an object that returns a new item using next(), and raises a StopIteration when no items are left

```
iterator = iter([“Batman”, “Superman”, “Spiderman”])
print(next(iterator)) #Batman
print(next(iterator)) #Superman
print(next(iterator)) #Spiderman
print(next(iterator)) #Exception!

print(next(iterator), “END”) #avoids the exception, and displays END
```

Iterators

- To use your classes in for….in…. statements
  - An *iterator* (created with iter()) is an object that returns a new item using next(), and raises a StopIteration when no items are left

```
iterator = iter(["Batman", "Superman", "Spiderman"])

for fruit in iterator:
    print(fruit)
```

Iterators

- An object, to be used as an iterable, has to implement
  - __iter__() (that returns the object itself) and __next__()

```
class Fibo:
    def __init__(self):
        self.seq = [1, 1, 2, 3, 5, 8, 13, 21, 34, 55]
    def __iter__(self):
        return self
    def __next__(self):
        if len(self.seq) > 0:
            return self.seq.pop(0)
        raise StopIteration()

fseq = Fibo()
for n in fseq:
    print(n, end = " ")
```

First way:

a container

Iterators

- An object, to be used as an iterable, has to implement
  - __iter__() (that returns the object itself) and __next__()

```
class Fibo:
    def __init__(self):
        self.seq = [1, 1, 2, 3, 5, 8, 13, 21, 34, 55]
        self.index = -1
    def __iter__(self):
        return self
    def __next__(self):
        if self.index < len(self.seq) - 1:
            self.index += 1
            return self.seq[self.index]
        raise StopIteration()
    def reset(self):
        self.index = -1
fseq = Fibo()
for n in fseq:
    print(n, end = " ")
fseq.reset()
for n in fseq:
    print(n, end = " ")
```

Second way:

an index

Iterators

- An object, to be used as an iterable, has to implement
  - __iter__() (that returns the object itself) and __next__()

```
class Fibo:
    def reset(self):
        self.nr1 = 0
        self.nr2 = 1
    def __init__(self, maxnum=1000):
        self.maxnum = maxnum
        self.reset
    def __iter__(self):
        return self
    def __next__(self):
        if self.nr2 > self.maxnum:
            raise StopIteration()
        nr3 = self.nr1 + self.nr2
        self.nr1 = self.nr2
        self.nr2 = nr3
        return self.nr1
fseq = Fibo()
for n in fseq:
    print(n, end = " ")
```

Third way:

items generated

How many items

generated here?

Iterators

- An object, to be used as an iterable, has to implement
  - __iter__() (that returns the object itself) and __next__()

```
class Fibo:
    def reset(self):
        self.nr1 = 0
        self.nr2 = 1
    def __init__(self, maxnum=1000):
        self.maxnum = maxnum
        self.reset
    def __iter__(self):
        return self
    def __next__(self):
        if self.nr2 < self.maxnum:
            raise StopIteration()
        nr3 = self.nr1 + self.nr2
        self.nr1 = self.nr2
        self.nr2 = nr3
        return self.nr1
fseq = Fibo()
for n in fseq:
    print(n, end = " ")
```

Third way:

items generated

the length of the Fibonacci sequence up to maxnum

Warning with potentially infinite iterables!

Generators

- Lighter way to iterate over a number of generated elements
  - When __next__() is called on a generator, the function is executed until statement yield is reached
  - Then it waits, until __next__() is called again

```
def fibo(maxnum):
        nr1 = 0
        nr2 = 1
        while nr2 <= maxnum:
            nr3 = nr1 + nr2
            nr1 = nr2
            nr2 = nr3
            yield nr1

fseq = fibo(1000)
for n in fseq:
    print(n, end = " ")
```

A function is a generator

just because contains yield

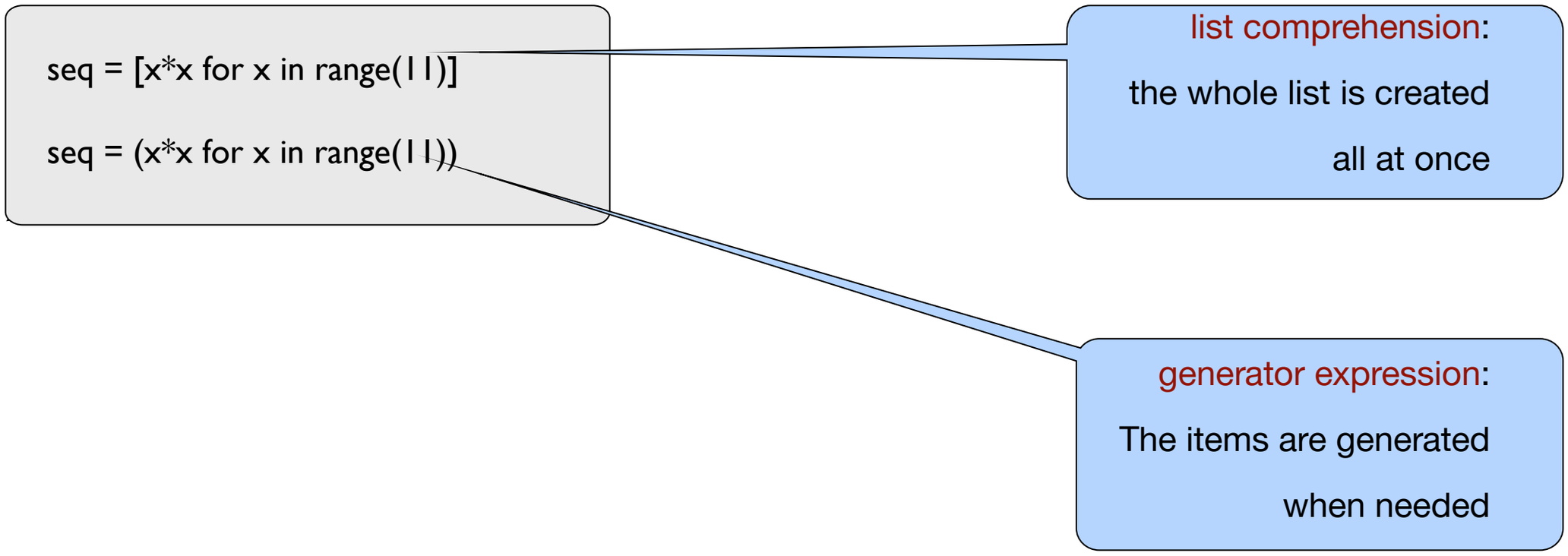
An iterable is automatically

created by Python

(with __next__() and __iter__())

Generator expression

- List comprehension vs Generated expression



Exercises

- **All** exercises in Chapter 23 of the reference book