Object-Oriented Programming in Python

Programmazione Avanzata 2025-26

Class

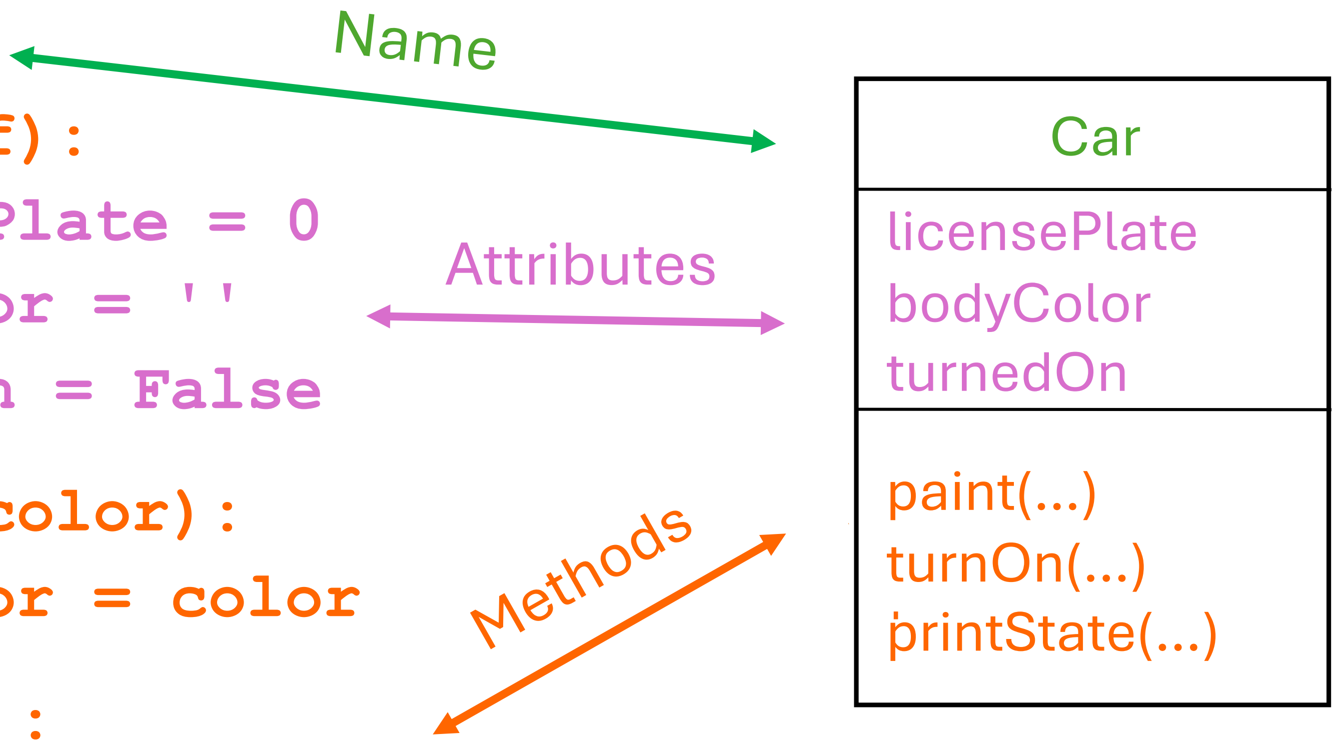
- Defined by the developer (e.g., **`Car`**) or Python itself (e.g., **`File`**)
- The instruction **`e = Car()`**
  - Allocates memory for the variable **`e`**, that acts as a "pointer" (reference) to the object
  - And invokes function **`__init()__`** to initialize the object, allocating its (instance) attributes and assigning a value to them

Class

- Object descriptor
  - Defines the common structure of a set of objects
- It consists of a set of members
  - Attributes, or properties
  - Methods, or operations

Class

```
class Car:
    def __init__(self):
        self.licensePlate = 0
        self.bodyColor = ''
        self.turnedOn = False

    def paint(self, color):
        self.bodyColor = color

    def turn_on(self):
        self.turned_on = True

    def printCar(self):
        print(f"Plate: {licensePlate}, color: {bodyColor}")
```



Methods

- Methods represent messages that an object can accept:
  - `turnOn`
  - `paint`
  - `printState`
- Methods may have parameters (may accept arguments)
  - `paint('red')`
- In a method, there can be only one method with a given name (no duplicates)
  - Difference between Python and other languages, like Java, where methods overloading, i.e., having methods with same name but different parameters is allowed, to enhance flexibility

Current object, a.k.a. self

- By convention, each method receives, as a first parameter, the reference to the object instance
- By convention, this parameter is called **`self`**
- Upon calling a method, **`self`** is initialized with the reference to the instance
  - Instruction **`c.turnOn()`** sets **`self`** to **`c`**
  - Equivalent to **`Car.turnOn(c)`**
- The use of **`self`** allows to avoid name ambiguities
- Using **`self`** is mandatory for referring to current object

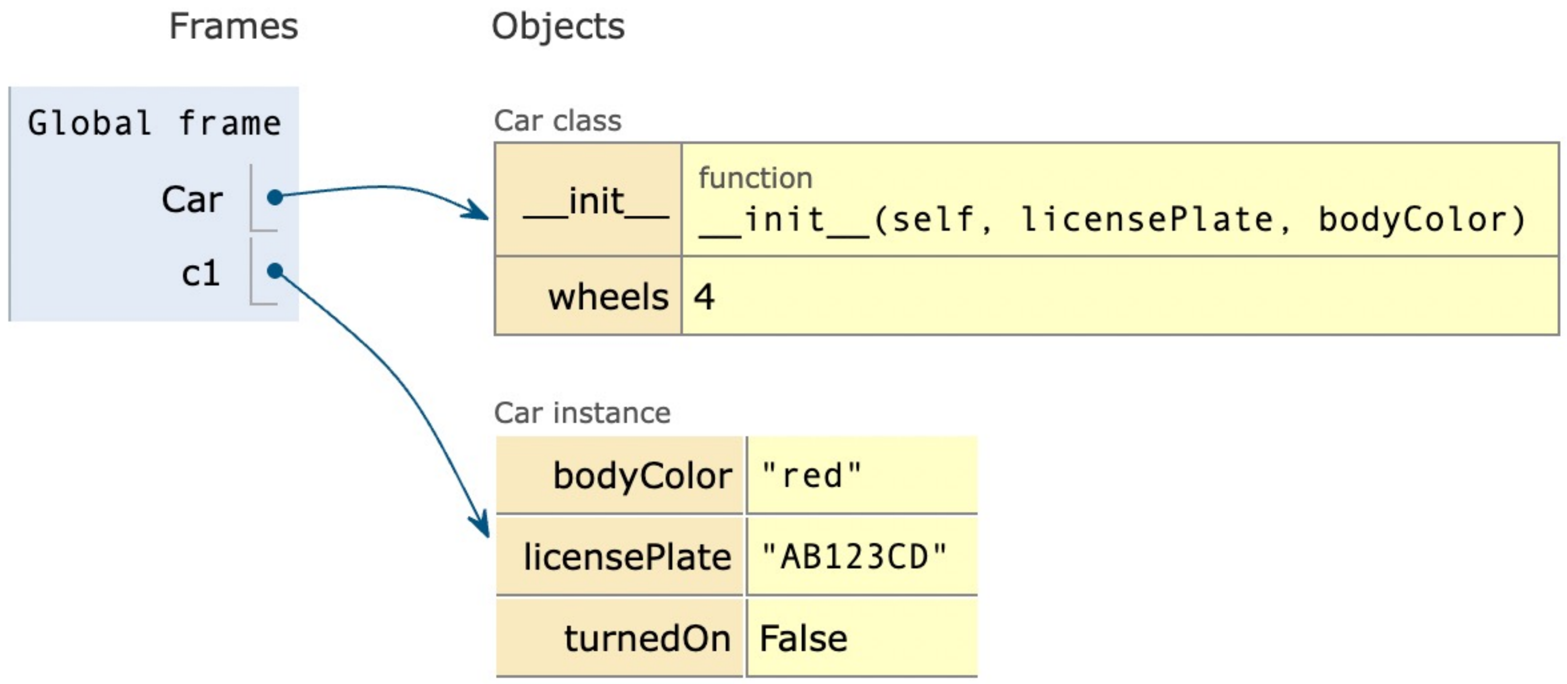
Class attributes vs instance attributes

- Instance attributes are defined in the `__init()__` method and assume a specific value for the particular object (instance)
- Class attributes, instead, are shared among all instances
  - Can be modified accessing them using the class name
  - If the instance name is used, a local (instance) variable is created

Class attributes vs instance attributes

```
class Car:

    wheels = 4 # class attribute

    def __init__(self, licensePlate, bodyColor):
        self.licensePlate = licensePlate # instance attr.
        self.bodyColor = bodyColor # instance attr.

print(Car.wheels) # prints 4
c1 = Car('AB123CD','red') # print(c1.wheels) also prints 4
c2 = Car(...)
c2.wheels = 6 # separate instance variable created in c2
print(c1.wheels) # prints 4
Car.wheels = 6 # class attribute can be changed w/ class name
```

Class attributes vs instance attributes



Dynamic nature of attributes

- Instance attributes are normally defined in **`__init()__`**
  - All instances will have the same set of attributes
  - Their value may be redefined in methods (**`self.name`**) or in external code (**`c1.name`**)
- However, new attributes may be created later
  - In any instance method (just assign a value to **`self.newName`**)
  - In the external code (just assign a value to **`c1.newName`**)
  - Such attribute is assigned to that specific instance, only
  - Works also for class-level attributes (**`Car.newName`**)
  - Avoid as much as possible, as code gets much less readable

Objects

- An object is identified by:
  - Its class, which defines its structure (attributes and methods)
  - Its state (values of attributes)
  - An internal unique identier

Objects creation

- Creation of an object is achieved using the name of the class, followed by **`()`**

```
c = Car()
```

- Python calls special method **`__init()__`**, if defined within the class
- Arguments can be passed to initialize the object being created

Objects creation

```
class Car:
    def __init__(self, licensePlate, bodyColor):
        self.licensePlate = licensePlate
        self.body_color = bodyColor
        self.turned_on = False

    def paint(self, color):
        self.bodyColor = color

    def turnOn(self):
        self.turnedOn = True

c = Car('XX777YY', 'white')
# c.licensePlate = 'AB123CD'
c.paint('red')
c.turnOn()
```

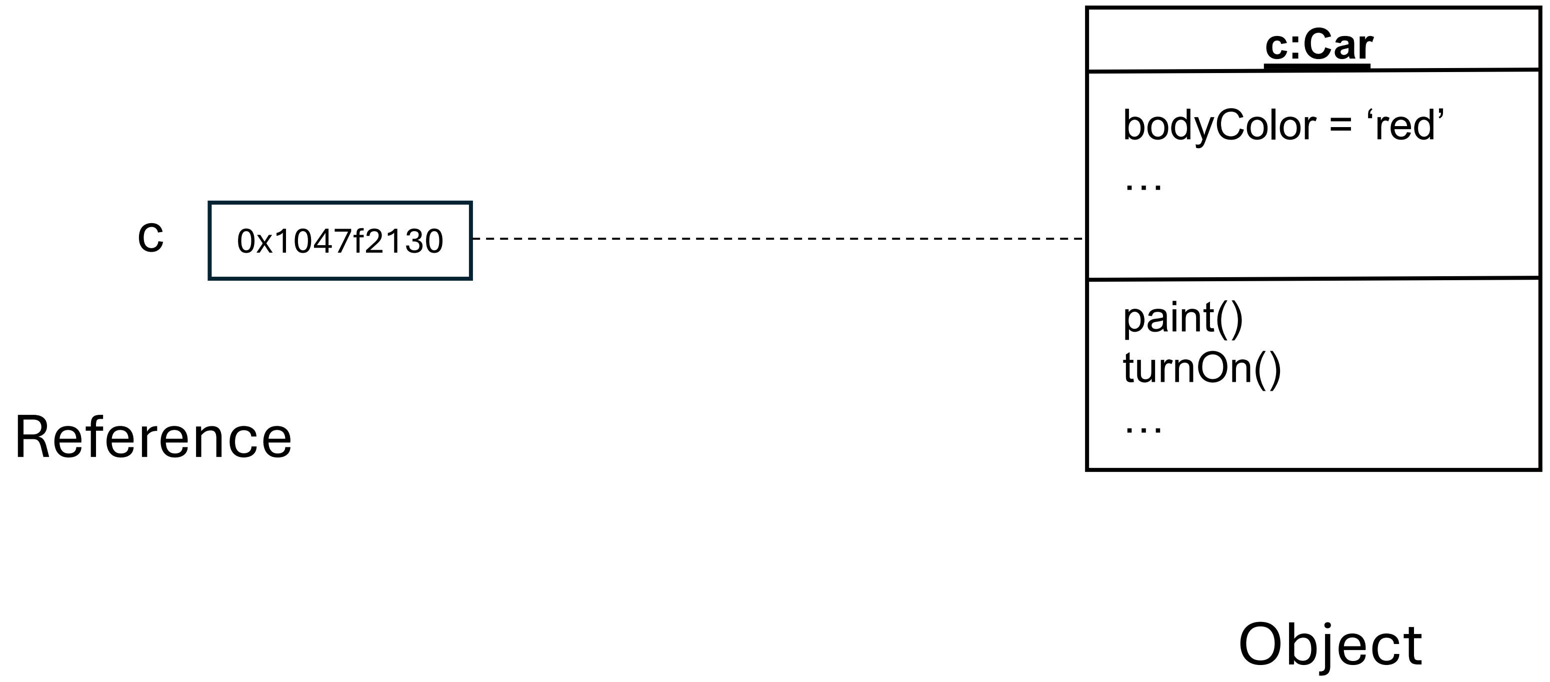
Objects and references

- Object creation returns a so called reference to the piece of memory containing the created class instance
- By invoking `print()` on an object name whose class is `Car`, its memory address is printed, not its content

```
<__main__.Car object at 0x1047f2130>
```

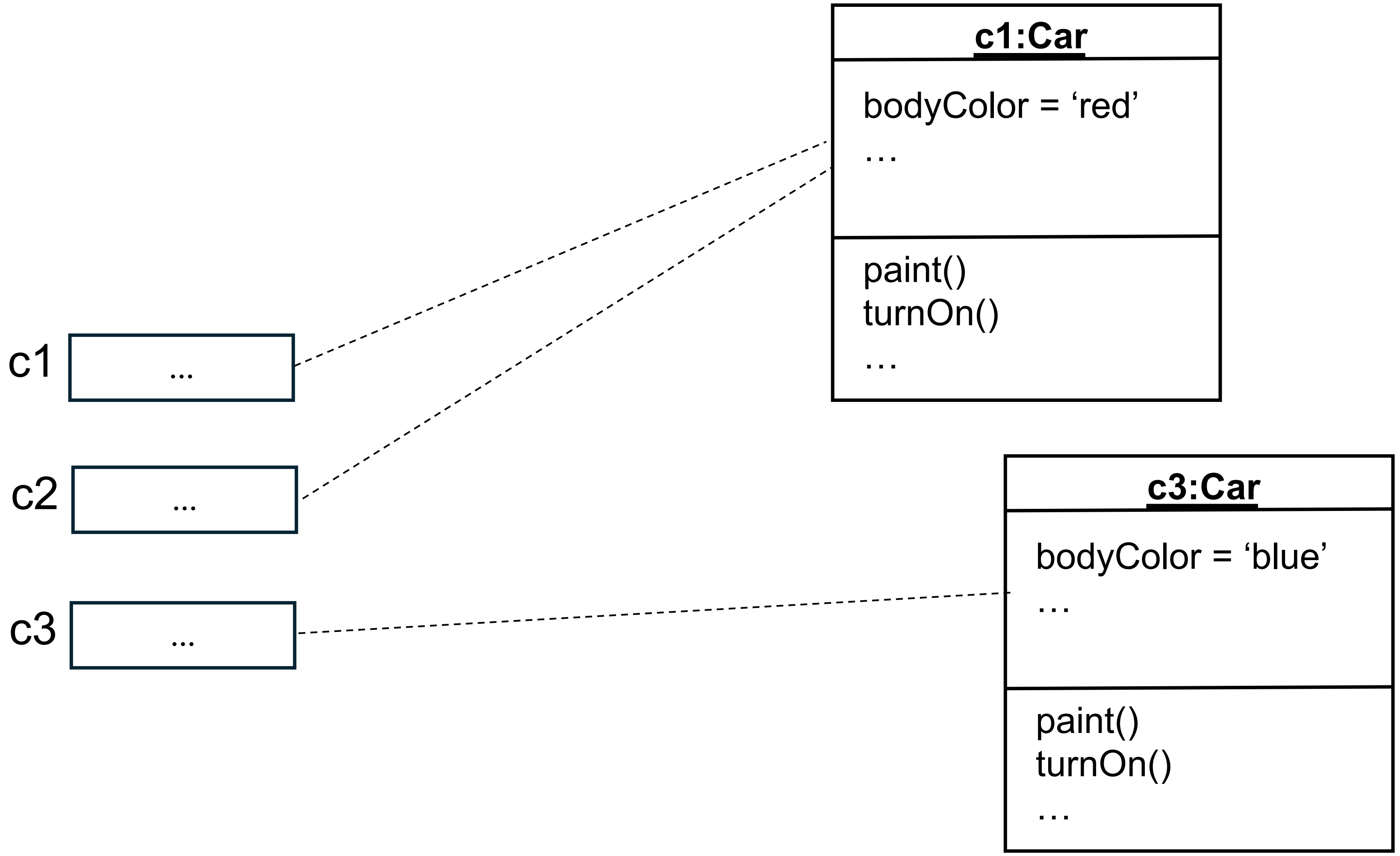
Aliasing

- An object can have, at some point zero, one or more references (*names*) pointing to it



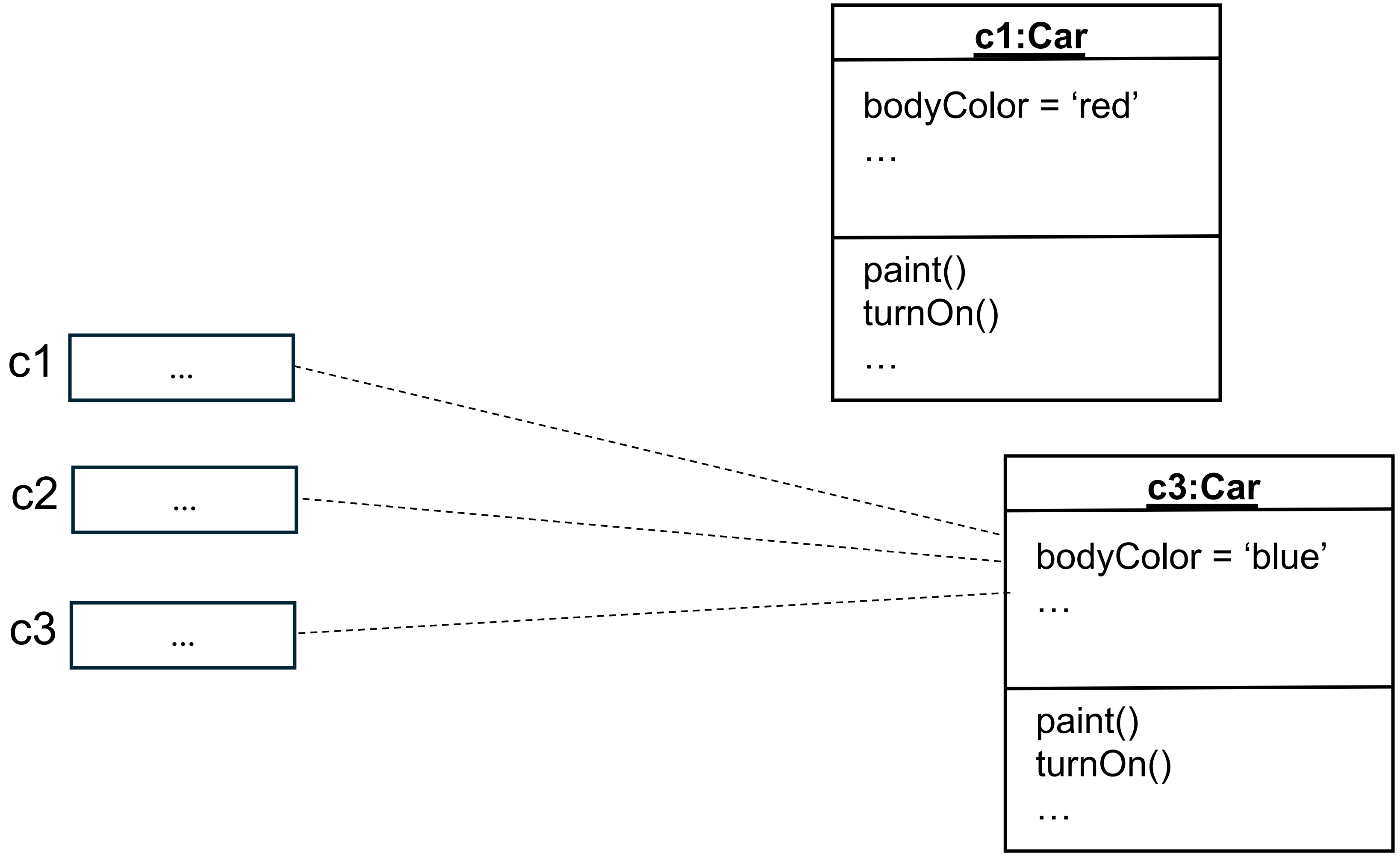
Aliasing

```
c1 = Car()
c2 = c1
c3 = Car()
```



Aliasing

```
c1 = Car()
c2 = c1
c3 = Car()
c1 = c3
c1 = c3
```



Heap

- It is a part of the memory used by an executing program to store data dynamically created at run-time
- In languages like, e.g., C:
  - Use of functions like **`malloc()`**, **`calloc()`** and **`free()`**
  - Instances of types in static memory or in heap
- In Python
  - Instances (objects) are always in the heap

Objects destruction

- In Python, memory release, differently than in other languages like C, is not a programmer’s concern
  - Managed memory language, based on a garbage collector
- Before the object is really destroyed (when all references to the object have been deleted), the method `__del()__`, if defined in the class, is invoked

Current object, a.k.a. self

- During the execution of a method, to refer to the current object **`self`** is used

```
class Book:
    def __init__(self, pages):
         self.pages = pages
    def readPage(self, n):
        print(f"Reading page {n}")
    def readAll(self):
        for i in range(1, self.pages):
            self.readPage(i)
```

Method invocation

- A method is invoked using dot or dotted notation

```
objectReference.method(arguments)
```

- Example:

```
c = Car()
c.paint("blue")
c.turnOn()
```

Access to attributes

- Attributes can be accessed using the dot notation too

```
objectReference.attribute
```

- Dot notations can be combined (chained), both for attributes and method invocations

Scope and encapsulation

- Example
  - Design the user interface for a laundry machine

Scope and encapsulation

- Design 1
  - Commands: time, temperature, amount of soap
  - Different values depending if one washes cotton, wool, ….
- Design 2
  - Commands: key C for cotton, W for wool, Key D for knitted robes
- Design 3
  - Command: wash
  - Insert clothes, and the washing machine automatically selects the correct program

Scope and encapsulation

- There are different solutions with different levels of granularity / abstraction

Motivation

- Modularity = cut-down inter-components interaction
- Information hiding = identifying and delegating responsibilities to components
  - Components = classes
  - Interaction = read/write attributes
  - Interaction = calling a method
- Heuristics
  - Attributes invisible outside the class
  - Visible methods are the ones that can be invoked from outside the class

Scope and syntax

- Variables are generally visible and accessible in the block of code they have been defined into

Visibility convention and modifiers

- OO programming languages often envisage methods to modify the visibility of class members
- Typically, at least private and public visibility are supported
  - Private attributes/methods are visible and accessible from instances of the same class only
  - Public attributes/methods are visible and accessible from everywhere

Visibility convention and modifiers

- In Python, all class-level attributes and instance-level attributes are public, i.e., they are always visible
  - Just read/write the attribute value as needed

Visibility convention and modifiers

- By convention, if the attribute is to be considered as private, its name can be prefixed with one `_` or two `__` underscores
  - **`self.counter`**
    - May be accessed (read/written) by anyone
  - **`self._counter`**
    - May still be accessed by anyone with `_`, but it is not polite to do that, and the IDE may raise a warning: hence, it should be regarded as private
  - **`self.__counter`**
    - It is difficult to access it from outside a method (Python will mangle its name to **`__ClassName__counter`**), so it could not be accessed by mistake (unless one really really wants to)

Getter and setter methods

- In other OO programming languages, the convention is to block/discourage direct access to attributes via dot notation
- Rather, the creation of so called getter and setter methods is typically recommended
  - Access to attributes, typically declared as private, is mediated by (public) getter/setter methods, with the same name of the attributes
  - Besides forcing the developer to think about which attributes can/shall be made accessible, in the methods, additional control code can be added

Getter and setter methods

- This behavior, which may be helpful sometimes, can be achieved also in Python using
  - The `@property` annotation for the getter method
  - The `@name.setter` annotation for the setter method (if omitted, the attribute will be read only)
- Both methods have the name of the attribute/property
  - Trasparent for the programmer, who can continue accessing the attribute remaining unaware of whether the attribute was meant to be hidden or not

Getter and setter methods

```
class Car:
    def __init__(self, licensePlate, bodyColor):
        self.licensePlate = licensePlate
        self.bodyColor = bodyColor
        self.turnedOn = False
        self._voltage = 12
    @property
    def voltage(self):
        return self._voltage
    @voltage.setter
    def voltage(self, volts):
        print('Warning: this can cause problems')
        self._voltage = volts
```

Special methods

- All objects can customize some of their behaviors by defining special methods
- Such methods have all a double underscore at the beginning and end of the name
- Hence, the definition of “dunder” methods
- Example: **`__init__(self, ...)`**, pronounced dunder-init

Dunder methods: convert to string

- To produce a string-printable representation of an object it is possible to use **`__str__(self)`**

```
class Car:
    ...
    def __str__(self):
        return f'{self.licensePlate}
                 {self.bodyColor}
                 {self.turnedOn}'

c1 = Car('AB123CD','red')
print(c1) # equivalent to print(str(c1))
```

```
>>> AB123CD red False
```

Dunder methods: convert to string

- To produce a programmer-oriented, string-printable representation, instead, **`__repr__(self)`** can be used

```
class Car:
    ...
     def __repr__(self):
         return f'{type(self).__name__} # class name
                  (licensePlate={self.licensePlate},
                  bodyColor={self.bodyColor},
                  turnedOn={self.turnedOn})'

c1 = Car('AB123CD','red')
print(repr(c1))

>>> Car (licensePlate=AB123CD, bodyColor=red, turnedOn=False
```

Inheritance

- A class can be a sub-type of another class
- The derived class
  - Implicitly contains (inherits) all the members of the class it inherits from
  - Plus, it can augment its structure with any additional member that it defines explicitly
  - Can also override (change) the definition of existing methods by providing its own implementation
- The code of the derived class consists only of changes and additions to the base class

Addition

```
class Employee:
    def __init__(self, name, wage):
        self.name = name
        self.wage = wage

    def increment_wage(self):
        self.wage += 5000

class Manager(Employee):
    def __init__(self, name, wage, managedUnit):
        super().__init__(name, wage)
        self.managedUnit = managedUnit

m = Manager('The Manager', 100000, 'HR')
m.increment_wage() # OK, inherited, a Manager is an Employee
>>> 105000
```

Overriding

```
class Employee:
    …
    def increment_wage(self):
        self.wage += 5000

class Manager(Employee):
    def __init__(self, name, wage, managedUnit):
        super().__init__(name, wage)
        self.managedUnit = managedUnit
    def increment_wage(self): # overrides that in Employee
        self.wage += 25000

m = Manager('The Manager', 100000, 'HR')
m.increment_wage() # adds 25000 to m.wage rather than 5000
>>> 125000
```

Super (reference)

- As much like **`this`** is a reference to the current object, **`super`** is used to refer to parent class

Inheritance and objects construction

- An object of a child class “contains” an instance of the parent class
  - Hence, the latter must be initialized by calling parent class **`__init()__`** method with **`super().`** or **`ClassName.`** passing parameters as needed
  - The call needs to be inserted as first statement of each child **`__init()__`** method
  - Construction proceeds top-down in the inheritance hierarchy

Why inheritance

- Frequently, a class is merely a modification of another class: in this way, inheritance helps minimize repetition of the same code
- Localization of code
  - Fixing a bug in the base class automatically fixes it in the subclasses
  - Adding functionality in the base class automatically adds it in the subclasses too
  - Less chances of different (and inconsistent) implementations of the same operation

Polymorphism

- Polymorphism in object-oriented programming refers to the ability of different objects to respond to the same message (method call) in different ways
  - It allows one interface (set of methods) to be used for a general class of actions, with the specific action determined by the exact nature (class) of the object that is using it
  - In other words, the same method name to behave differently based on the object that is calling it
- It enables code reusability, flexibility, and simplifies maintenance

Polymorphism

- There are two types of polymorphism (each referred to with different names depending on the programming language being considered)
  - Method overriding (a.k.a. runtime polymorphism)
  - Duck typing (or informal polymorphism)

Runtime polymorphism

- A child class overrides a method from its parent class
- The actual method to call is determined dynamically at runtime using dynamic method dispatch (hence the name, runtime polymorphism)
- Several objects from different classes, sharing a common ancestor class can be treated uniformly
- Algorithms can be written for the base class (using the relative methods) and applied to any subclass

Runtime polymorphism

```
e = Employee('The employee', 20000)
m = Manager('The Manager', 100000, 'HR')

employees = [e, m] # storing both employees and managers

for employee in employees:
    employee.increment_wage() # most specific version called

for employee in employees:
    print (f"{employee.name} {employee.wage}")

>>> The employee 25000
>>> The Manager 150000
```

Runtime polymorphism

```
class Animal:
    def speak(self):
        print("Animal makes a sound")

class Dog(Animal):
    def speak(self):
        print("Dog barks")

class Cat(Animal):
    def speak(self):
        print("Cat meows")

animals = [Dog(), Cat(), Animal()]
for a in animals:
    a.speak()
```

```
>>> Dog barks Cat meows Animal makes a sound
```

Duck typing

- Ispired by saying “If it walks like a duck and it quacks like a duck, then it must be a duck”, meaning that if something behaves like a particular thing, probably is that thing
- If an object behaves like a certain type, it can be used as such even though it is not that type
  - In practice, even though two classes do not inherit from the same base class, if they both implement a given method, they can be passed to a function that calls that method

Duck typing

- Python, differently than other languages, focuses on behavior, not inheritance hierarchy
  - The type or the class of an object is less important than the methods it defines
  - When using duck typing, types are not checked: instead, the presence of a given method or attribute matters
- Algorithms can be written assuming the existence of a common behavior (based on a common methods) and applied to different classes

Duck typing

```
class Cat: # not inheriting from Animal in this case
    def speak(self):
        print("Meow")

class Dog: # not inheriting from Animal in this case
    def speak(self):
        print("Woof")

def animal_sound(animal):
    animal.speak()

cat = Cat()
dog = Dog()

animal_sound(cat)  # outputs Meow, works because Cat defines speak()
animal_sound(dog)  # outputs Woof, works because Dog defines speak()
```

Duck typing

- Considering for instance the following code

```
def pretty_print(data_provider):
    data = data_provider.read_data()
    for d in data:
  print(d[0])
```

- What is the allowed type of data_provider?
    - Duck typing says any class that has a **`read_data()`** method

Duck typing

- The function **`pretty_print()`** may be called with totally different, unrelated classes as parameters

```
# on a database
source_database = DatabaseAccess('localhost’,
                                 'root', 'root', 'data')
pretty_print(source_database)

# on a csv file
source_file = FileAccess('data.csv')
pretty_print(source_file)
```

Checking class type

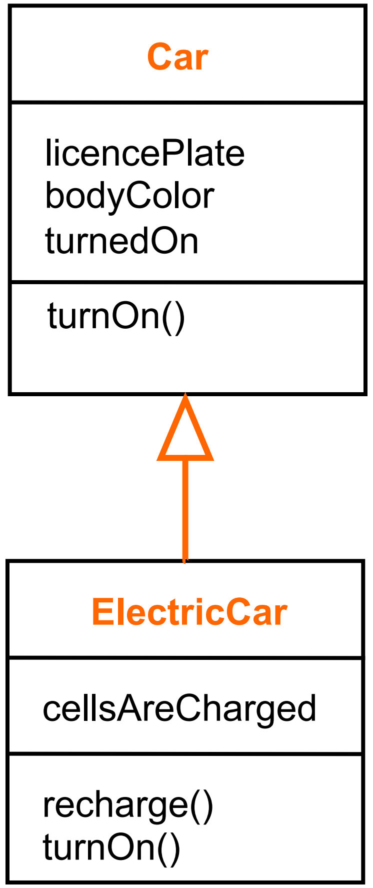
- Inside a polymorphic function, it is possible to check the classes of the received instances using **`isinstance()`**
- Useful to avoid errors before calling methods that might not exist
- Abusing of this checks should be avoided, as they defeat the simplicity of duck typing mechanism

Protocols

- Many built-in functions, operators, and keywords are polymorphic
- The set of required methods is called protocol
- Examples:
  - The **`len()`** function accepts any object with a **`__len__()`** method
  - Any object can be iterated if it has a **`__iter__()`** method
  - An object can be indexed if it has a **`__getitem__()`** method
  - An object may be used in the **`with`** statement if it implements an **`__enter__()`** and an **`__exit__()`** method

Inheritance in few words

```
class Car:

    def __init__(self, licensePlate, bodyColor):
        self.licensePlate = licensePlate
        self.bodyColor = bodyColor
        self.turnedOn = False

    def turnOn(self):
        self.turnedOn = True

class ElectricCar(Car):

    def __init__(self, licensePlate, bodyColor):
        super().__init__(licensePlate, bodyColor) # or Car.
        self.cellAreCharged = False

    def recharge(self):
        self.cellAreCharged = True

    def turnOn(self):
        if(self.cellAreCharged):
            self.turnedOn = True
```



Inheritance in few words

- Subclass
  - Inherits attributes and methods defined in the base classes
  - Can add new attributes and methods
  - Can modify inherited attributes and methods (override)

Inheritance in few words

- Class **`ElectricCar`** inherits
  - Attributes: **`licencePlate`**, **`bodyColor`**, **`turnedOn`**,
  - Method: **`paint()`**
- Modifies (overrides)
  - Method: **`turnOn()`**
- Adds
  - Attribute: **`cellsAreCharged`**
  - Method: **`recharge()`**

Inheritance in few words

```
class Car():
      def turnOn():
            …
Class ElectricCar(Car):
      def turnOn():
            …

c1 = Car('AB123CD', 'red')
c2 = ElectricCar('EF456GH', 'blue')
c3 = Car('IJ789KL', 'black')

garage = [c1, c2, c3]
for c in garage:
    c.turnOn()
```

Multi-level inheritance

- Inheritance can be applied at multiple stages

```
class B(A):
   ...
class C(B):
   ...
class D(B):
   ...
```

Multiple inheritance

- In Python, differently than in other OO programming languages like, e.g., Java, it is possible for a class to inherit from more than one superclass

```
Class SportsCar(Car, ExpGadget)
```

- All attributes and methods for both superclasses are inherited, in the order of declaration
- In the subclass, both the constructors must be called, **`Car.__init__()`** and **`ExpGadget. __init__()`**

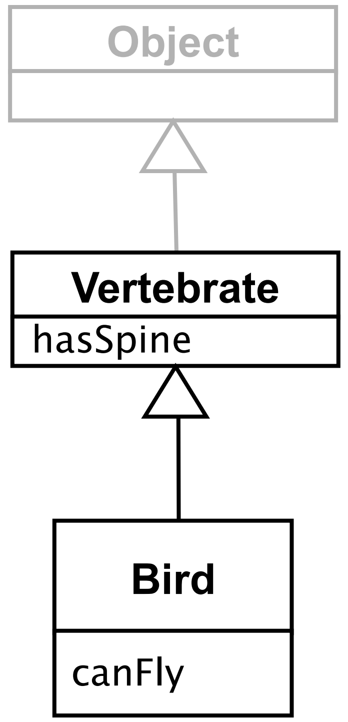
Object

- All classes are subtypes of **`Object`**

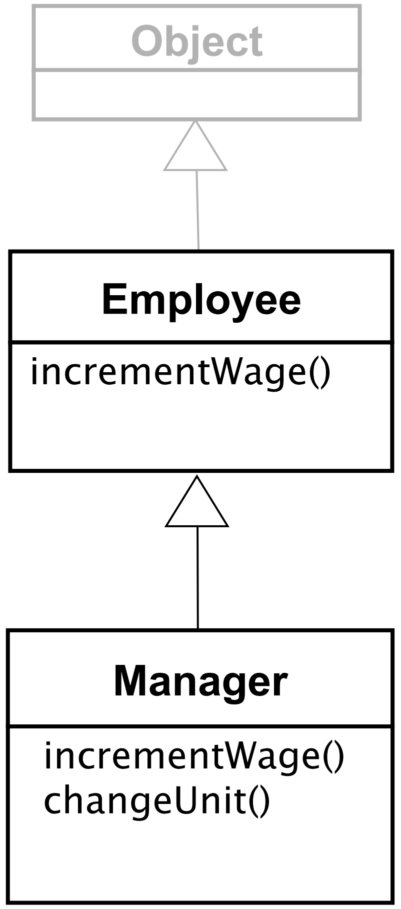
```
class Vertebrate:
    ...
class Bird(Vertebrate):
    ...
```

- Implicitly, it is

```
class Vertebrate(Object)
    ...
```



Object


Object
Employee
incrementWage()
Manager
incrementWage()
changeUnit()

Why a class Object

- Any instance of any class can be seen as an instance of class `Object`
- `Object` class
  - Defines some common operations, which are useful for (and are inherited by) all classes
  - Often, they are overridden in sub-classes

Methods of class Object

```
__init__()
__str__()
__repr__()
__eq__()
__hash__()
__getattribute__()
__setattr__()
__class__
...
```

Printing an object

- When calling, e.g., **`print(obj)`** on an object of a given class, Python internally call **`print(str(obj))`**
  - If an **`__str__()`** method has not been defined in that given class, Python uses the default behavior from the base **`Object`** class
  - The default implementation in the **`Object`** class returns **`<__main__.ClassName object at memory_address>`**
- To make **`print(obj)`** return something meaningful, **`__str__()`** can (should) be overridden
- Recommended for many common methods in **`Object`**

Comparing two objects

- Another key method of class `Object` is `__eq__()`, which defines the behavior of the == operator
- Every class inherits a default version of `__eq__()` from the base `Object` class, unless it overrides it
- The predefined behavior of `__eq__()` in the `Object` class consists in comparing object identity, i.e., it returns `True` only if the two objects are the exact same instance in memory

Comparing two objects

```
class MyClass:
    pass

a = MyClass()
b = MyClass()
c = a

print(a == b)  # False (different instances)
print(a == c)  # True  (same instance)
```

Comparing two objects

- Overriding **`__eq__()`** make it possible that two different instances are be considered equal based on their contents (attributes) rather than their identities (i.e., memory address, default behavior)
- In this way, an overloaded version of **`==`** is defined
- When doing that, it may be helpful to use the built-in function **`isinstance(object, ClassName)`**, which allows to check whether an object is an instance of a particular class whose name is **`ClassName`**

Comparing two objects

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

    def __eq__(self, other):
        if isinstance(other, Person):
            return self.name == other.name and self.age == other.age
        return False

p1 = Person("Alice", 30)
p2 = Person("Alice", 30)
p3 = Person("Bob", 25)

print(p1 == p2)  # True (contents match, i.e., same class type and attributes)
print(p1 == p3)  # False
```

Dunder methods: comparisons

- Similarly to how the **`==`** can be redefined, to implement the **`<`** operator, the function to use is **`__lt__(self, other)`**
- This way, an overloaded version of **`<`** is obtained
- Other operators (**`>`**, **`<=`**, **`!=`**, **`>=`**) are inferred from these methods
- All data structures (dictionaries, sets, ...) and methods (**`sort`**, **`max`**, **`index`**, ...) honor these operators

Operators overloading

```
object.__add__(self, other)
object.__sub__(self, other)
object.__mul__(self, other)
object.__matmul__(self, other)
object.__truediv__(self, other)
object.__floordiv__(self, other)
object.__mod__(self, other)
object.__divmod__(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.__neg__(self)
object.__pos__(self)
object.__abs__(self)
object.__invert__(self)

object.__complex__(self)
object.__int__(self)
object.__float__(self)
```

```
object.__radd__(self, other)
object.__rsub__(self, other)
object.__rmul__(self, other)
object.__rmatmul__(self, other)
object.__rtruediv__(self, other)
object.__rfloordiv__(self, other)
object.__rmod__(self, other)
object.__rdivmod__(self, other)
object.__rpow__(self, other[, modulo])
object.__rlshift__(self, other)
object.__rrshift__(self, other)
object.__rand__(self, other)
object.__rxor__(self, other)
object.__ror__(self, other)

object.__round__(self[, ndigits])
object.__trunc__(self)
object.__floor__(self)
object.__ceil__(self)

object.__index__(self)
```

A well defined class

- A proper Pyton class shall define ("boilerplate" code)
  - An **`__init__()`** method
  - A set of **`self.attribute`** attributes initialized in **`__init__()`**
  - A **`__repr__()`** method for conversion to a (programmer-, debug-oriented) string
  - An **`__eq__()`** method for allowing **`==`** and **`!=`** comparisons
  - If required, ordering methods such as **`__le__()`** for allowing **`<`**, **`>`**, **`<=`**, and **`>=`** comparisons
  - A **`__hash__()`** method to be used by sets and dict keys
  - If required, setter/getter methods for attributes
  - Plus any other methods specifying its behavior

Dataclasses

- The “boilerplate” code can be automatically generated by the **`@dataclass`** decorator
- Especially useful for classes with basic bahavior, such as “data container” classes

```
Class RegularCard:
    def __init__(self, rank, suit):
            self.rank = rank
            self.suit = suit
#plus boiler plate dunder methods
```

```
from data classes import dataclass

@dataclass
Class DataClassCard
        rank: int
        suit: str
```