



# Object Oriented Programming in Python

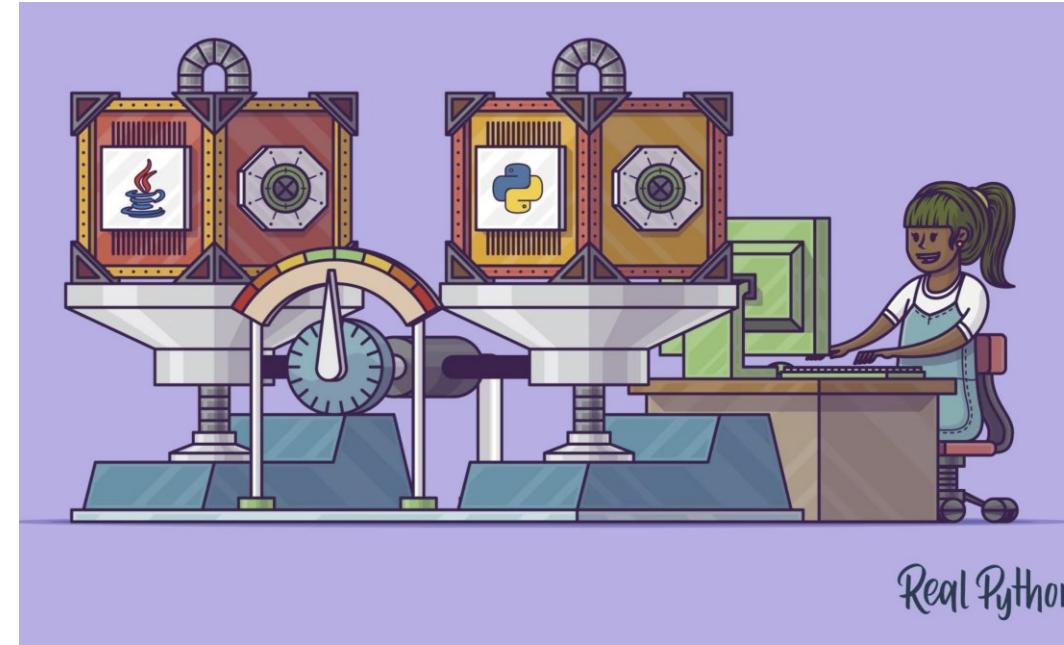
**Transitioning from Java to Python**

Fulvio Corno

Giuseppe Averta

Carlo Masone

Francesca Pistilli



Real Python

<https://realpython.com/oop-in-python-vs-java/>

<https://realpython.com/python3-object-oriented-programming/>

# Known OOP features (in Java)

- Classes
- Objects
- Properties
- Methods
- Visibility
- Constructor
- Encapsulation
- Inheritance
- Polymorphism
- Annotations
- Overloading

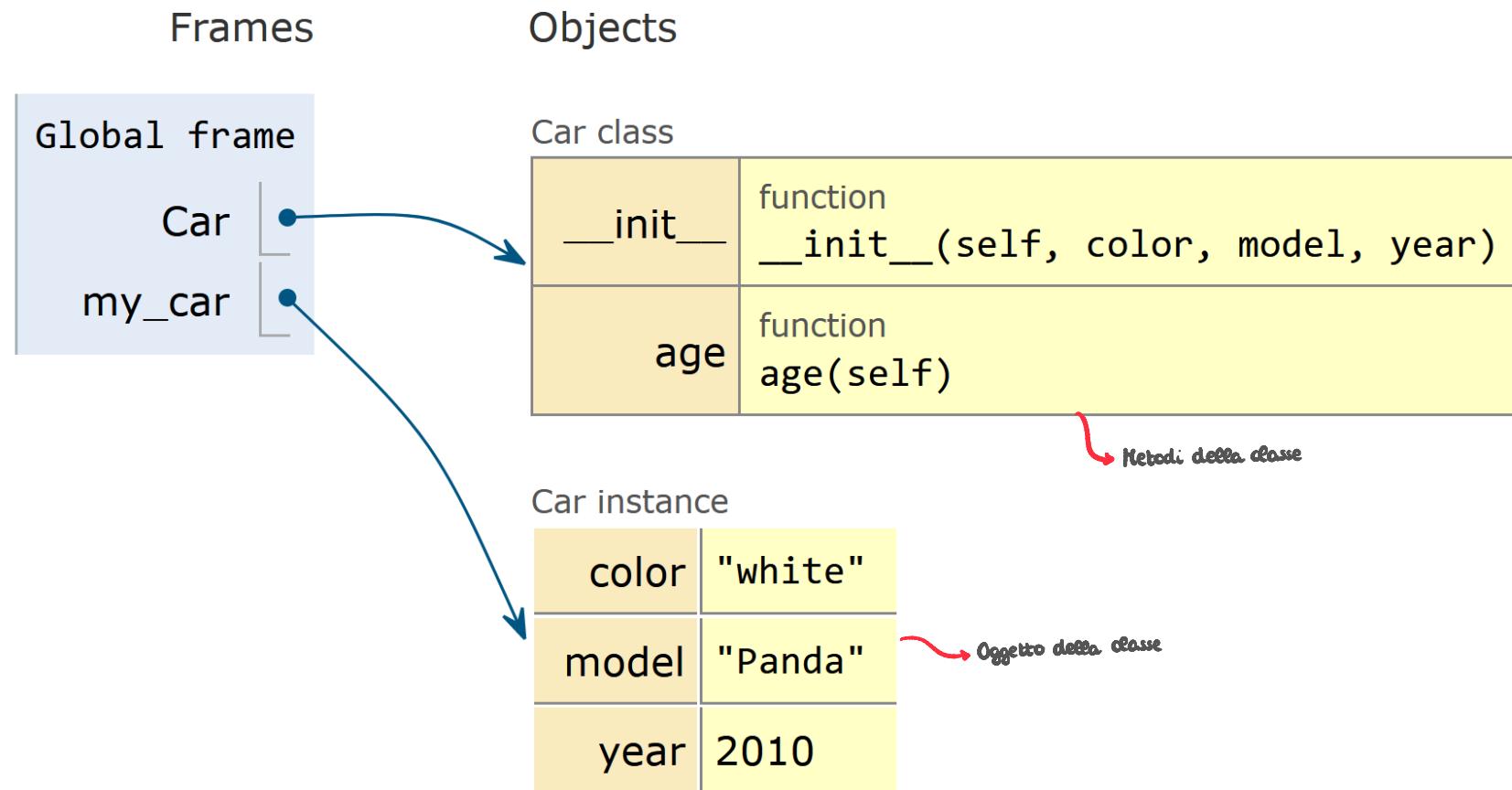
N.B.: In Python, qualunque valore è un oggetto  
a differenza di Java

We assume these concepts are known in  
Java, let's see how they map in Python

# Classes and Objects

```
class usato a creare  
una classe!  
Classe in Java!  
Anche il costruttore riceve SEMPRE  
in input self OBBLIGATORIO!  
  
class Car:  
    def __init__(self, color, model, year):  
        self.color = color  
        self.model = model  
        self.year = year  
  
    Come anche il costruttore,  
    tutti i metodi ricevono in  
    input come primo  
    parametro self.  
    def age(self):  
        return 2024 - self.year  
  
    self = this  
    In java this era facoltativo.  
    in python self è obbligatorio  
    e viene sempre utilizzato nella  
    definizione dei metodi all'interno  
    della classe!  
  
    } i metodi di questa classe li definiamo come facevamo con le funzioni; con def.  
  
my_car = Car("white", "Panda", 2010)  
  
print(my_car.age())  
  
Creazione di un oggetto della classe => non devo usare new.  
Passerò in input i valori degli attributi  
che definirò all'interno del costruttore!  
  
oggetto.metodo()
```

# Visual representation



# Classes and Objects

Class definition

```
class Car:  
    def __init__(self, color, model, year):  
        self.color = color  
        self.model = model  
        self.year = year
```

Method definition

```
def age(self):  
    return 2024 - self.year
```

Instance

```
my_car = Car("white", "Panda", 2010)
```

Method call

```
print(my_car.age())
```

# Classes and Objects

```
class Car:  
    def __init__(self, color, model, year):  
        self.color = color  
        self.model = model  
        self.year = year  
  
    def age(self):  
        return 2024 - self.year  
  
my_car = Car("white", "Panda", 2010)  
  
print(my_car.age())
```

Annotations in the code:

- The word `Constructor` is written in red above the class definition.
- The words `Constructor parameters` are written in red above the `__init__` method parameters.
- The words `Constructor arguments` are written in red above the `Car` constructor arguments.

# Classes and Objects

```
class Car:  
    def __init__(self, color, model, year):  
        self.color = color  
        self.model = model  
        self.year = year  
        Properties  
    def age(self):  
        return 2024 - self.year  
  
my_car = Car("white", "Panda", 2010)  
  
print(my_car.age())
```

# Classes and Objects

```
class Car:          "this"
    def __init__(self, color, model, year):
        self.color = color
        self.model = model
        self.year = year
                    "this"
    def age(self):      "this"
        return 2024 - self.year

my_car = Car("white", "Panda", 2010)
                    "this"
print(my_car.age())
```

# What is 'self'?

- Each method receives, as a **first** argument, 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
- `my_car.age()` sets **self** to `my_car`
  - Equivalent to `Car.age(my_car)` (static method call with explicit **self**)
- Using **self** is always mandatory (unlike `this`, that can be omitted)

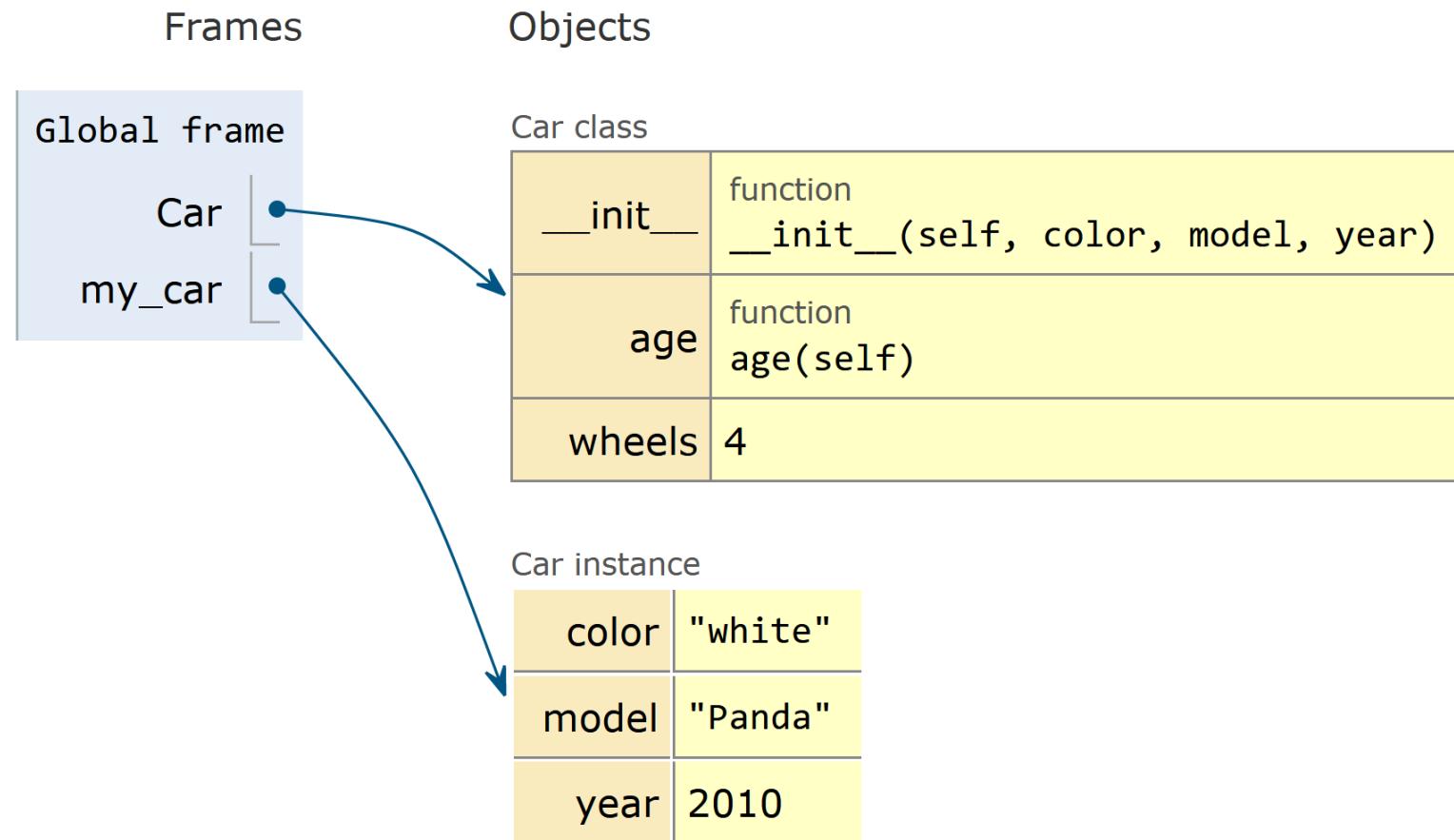


**self** è **SEMPRE** obbligatorio nella definizione dei metodi, => Un metodo, pur non ricevendo alcun parametro aggiuntivo in input, riceverà **SEMPRE** **self**.  
a differenza di `this` in java.

# Class attributes vs. instance attributes

```
class Car:  
    wheels = 4 # class attribute ('static' in Java)  
        ↳ Gli attributi che sono = a tutti gli oggetti della classe ('static') li definisco in questo modo!  
    def __init__(self, color, model, year):  
        self.color = color # instance attribute  
        self.model = model  
        self.year = year  
  
print(Car.wheels)  
print(my_car.wheels) # instances may access class attributes
```

# Class attributes vs. instance attributes



# Dynamic nature of attributes

- Instance attributes are normally **defined in the `__init__` constructor**
  - All instances will have the same set of attributes
  - Their **value** may be redefined in methods (`self.name`) or in external code (`my_car.name`)
- However, new attributes may be created later
  - In any instance method (just assign a value to `self.new_name`)
  - In the external code (just assign a value to `my_car.new_name`)
  - Such attribute is assigned to the specific instance, only
  - Works also for class-level attributes (`Car.new_name`)
  - Try to **avoid this possibility**, as it renders the code much less readable

Gli attributi di una classe vengono definiti all'interno del costruttore della classe stessa!  
Gli oggetti di una classe avranno medesimi attributi!

# Getters and Setters? No, thanks

- In Java, object properties (= instance attributes) are normally defined with a **private visibility**, and are not accessible from outside the class methods
  - `getXxx()` and `setXxx(xxx)` methods must be defined, for each property `xxx`
- In Python, attributes are **always visible**, and **no getter/setters** are required
  - Just read/write the attribute value

↳ In Python gli attributi sono sempre **public**! Quindi posso accedervi con la notazione puntata, senza l'utilizzo di getters/setters!

# Visibility conventions

- All class-level attributes and instance-level attributes are **public**
  - By **convention**, if you consider an attribute to be “private”, prefix it with one or two “\_” (underscore)
    - `self.counter`
      - may be accessed (read/written) by anyone
    - `self._counter` ↗ *Se c'è \_ prima dell'attributo, è come se il programmatore consigliasse all'utilizzatore di non accedere all'attributo, ma lui potrà ancora farlo!*
      - may still be accessed by anyone, but it's not polite to do that, and your IDE may send you a warning. You should consider it a private value
    - `self.__counter` (two underscores)
      - it is difficult to access if you are outside a method (Python will mangle its name to `_ClassName__counter`), so you will not access it by mistake (unless you really really want)
- Con \_\_ (doppio underscore), rendo ancora + difficile l'accesso a un attributo; rimane comunque ancora accessibile!*

# Getters and Setters, if/when you want them

- Need to customize what happens when you read/write a ‘private’ attribute?
- Use the `@property` annotation
- `@property` for the getter method
- `@name.setter` for the setter method
  - If omitted, it will be read-only
- Both methods have the `name` of the property

• Se voglio `getters|setters` in Python li posso avere;  
devo usare la notazione `@property` e `@name.setter`

```
1 class Car:  
2     def __init__(self, color, model, year):  
3         self.color = color  
4         self.model = model  
5         self.year = year  
6         self._voltage = 12  
7  
8     @property  
9     def voltage(self):  
10        return self._voltage  
11  
12    @voltage.setter  
13    def voltage(self, volts):  
14        print("Warning: this can cause problems!")  
15        self._voltage = volts
```

N.B.: x poter definire il setter DICO prima definire il getter (perché in `@name.setter`, name sarebbe il nome della property del getter) Mentre il getter posso definirlo anche senza definire il setter!

GETTER → il nome della funzione getter sarà = al nome dell'attributo senza \_ !

Net return mi devo ricordare di mettere \_ prima dell' attributo, essendo stato definito in questo modo.

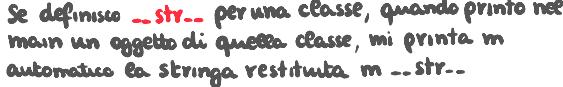
SETTER → Uso la notazione `@voltage.setter` voltage è il nome che ho dato al metodo GETTER!

# Special methods

- All objects can customize their behavior in implicit and arithmetic operators, by defining **special methods**
- Such methods have all a **double-underscore** at the **beginning & end** of the name
- Hence, the definition of “**dunder**” (double underscore) methods
- Example: `__init__(self, ...)` # pronounced: *dunder-init*
  - Full list of *dunder* methods:  
<https://docs.python.org/3/reference/datamodel.html#special-method-names>

• I metodi **dunder** sono i metodi definiti x la classe Object (da cui tutte le altre classi derivano) e che posso definire in override all' interno delle mie classi.  
Tutti questi metodi sono preceduti e seguiti da un **doppio underscore!**

# Dunder methods: convert to string

- \_\_str\_\_(self)  
  - string printable representation (like `toString()`)
- \_\_repr\_\_(self)
  - programmer-oriented printable representation (usually, the object creation)

```
class Car:  
    # ...  
  
    def __str__(self):  
        return f'{self.make}, {self.model}, {self.color}: ({self.year})'  
  
    def __repr__(self):  
        return (  
            f'{type(self).__name__}'  
            f'(make="{self.make}", '  
            f'model="{self.model}", '  
            f'year={self.year}, '  
            f'color="{self.color}")'  
)
```

Il metodo \_\_repr\_\_ mi restituisce una stringa contenente le info e i valori degli attributi dell'oggetto in modo standard, ovvero a come è stato creato l'oggetto.

```
>>> toyota_camry = Car("Toyota", "Camry", 2022, "Red")  
  
>>> str(toyota_camry)  
'Toyota, Camry, Red: (2022)'  
>>> print(toyota_camry)  
Toyota, Camry, Red: (2022)  
  
>>> toyota_camry  
Car(make="Toyota", model="Camry", year=2022, color="Red")  
>>> repr(toyota_camry)  
'Car(make="Toyota", model="Camry", year=2022, color="Red")'
```

# Dunder methods: comparisons

- `__eq__(self, other)`
  - implements `==` operator
  - Replaces Java's `.equal()`
- `__lt__(self, other)`
  - Implements `<` operator
  - Replaces Java's `Comparator`, `Comparable`, `compare()`, `compareTo()`
- Other operators (`>`, `<=`, `!=`, `>=`) are inferred from these methods
- All data structures (dictionaries, sets, ...) and methods (`sort`, `max`, `index`, ...) honor these operators

*equals()*

↳ Come in Java, dovrò definirlo in override nelle mie classi: se voglio confrontare due oggetti di questa classe sulla base di un attributo!

*compareTo() => Stesso discorso fatto x ...eq..!*

Se voglio usare questi metodi su strutture dati contenenti degli oggetti di una classe al loro interno, dovrò prima definire in override `__eq__` e `__lt__` nella classe!

# Dunder methods: 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)`
- Metodi **dunder** definiti su tutti gli oggetti (ovvero definiti sulla classe `Object`, da cui tutte le altre classi derivano)
- ↳ Tutti metodi che posso definire e override nelle mie classi!
- `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)`

# Inheritance

- A class may inherit from another class

- `class SportsCar(Car):`

Sportscar sarà **classe figlia** rispetto  
alla classe **Car**!

- All attributes and methods are inherited

- Must call parent class' `__init__` method

- `def __init__(self):`

- `Car.__init__()` # or: `super().__init__()`

- `self.speed = 'high'`

Eventuali attributi aggiuntivi della sottoclasse  
andranno definiti nel costruttore `__init__`!

Il costruttore della sottoclasse **DEVE** essere definito se  
è stato definito all'interno della classe padre!

Li potrà poi specializzare in overloading!

Con `super()`, richiamo un eventuale  
metodo della classe padre!

In questo caso devo aggiungerlo al costruttore della classe  
figlia x richiamare il costruttore della classe padre e definire,  
x un oggetto della classe figlia, anche gli attributi "base" della  
classe padre!

# Example

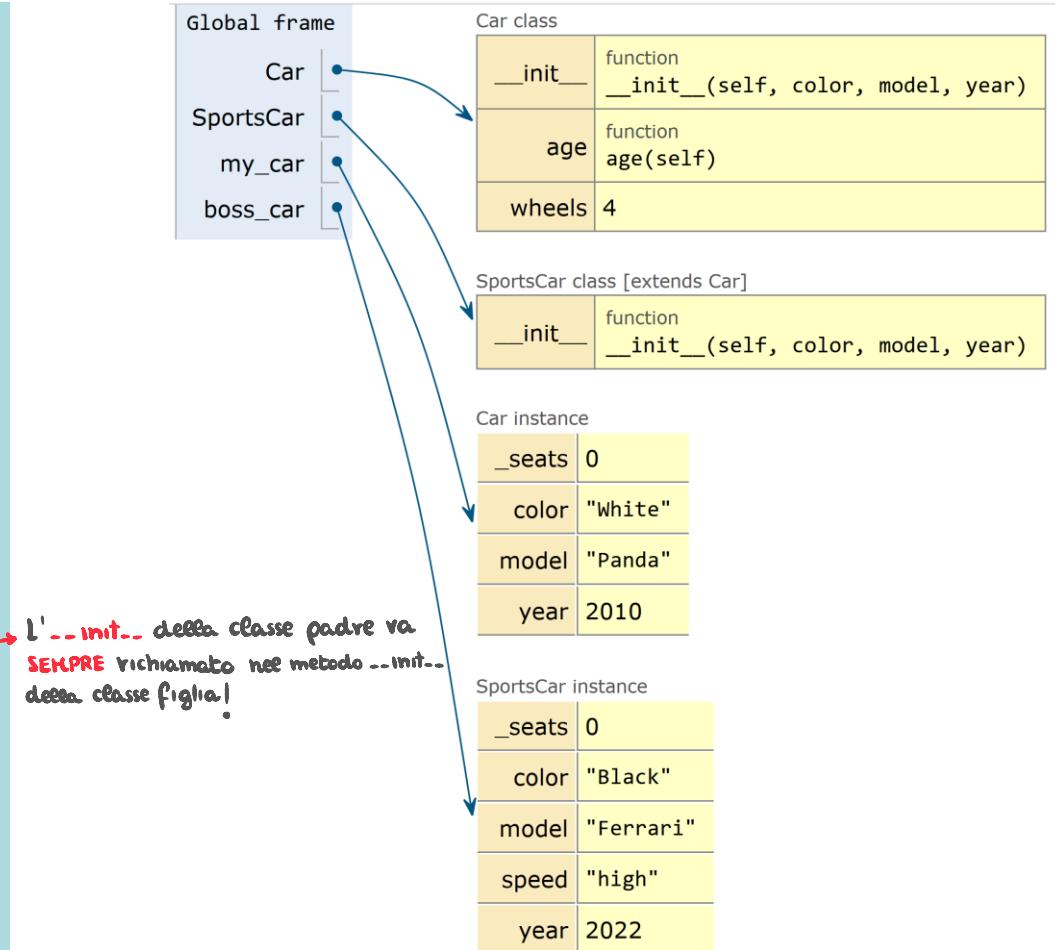
```
class Car:
    wheels = 4
    Definizione di una variabile globale
    Variabile CONDIVISA fra tutti gli oggetti
    della classe!
    • Corrisponde alle variabili static di java!

    def __init__(self, color, model, year):
        self.color = color
        self.model = model
        self.year = year
        self._seats = 0

    def age(self):
        return 2024 - self.year

class SportsCar(Car):
    def __init__(self, color, model, year):
        super().__init__(color, model, year)
        self.speed = 'high'

my_car = Car('White', 'Panda', 2010)
boss_car = SportsCar('Black', 'Ferrari', 2022)
```



# Multiple Inheritance

- In Python, una classe può essere sottoclasse di + classi!
  - ↳ In Java, questa cosa non è possibile!

- In Python, it's possible for a class to inherit from more than one superclass:

```
- class SportsCar(Car, ExpensiveGadget):
```

Metto tra parentesi tutte le classi di cui la classe che sto definendo sarà sottoclasse!

- All attributes and methods for both superclasses are imported, in the order of declaration ↳ La sottoclasse eredita metodi e attributi di tutte le super classi!

- Must call **both** constructors, Car.`__init__()` and ExpensiveGadget.`__init__()`

↳ Nell'`__init__` della sottoclasse, dovrò chiamare esplicitamente gli `__init__` di tutte le classi padre!

- There are no 'interfaces' in Python, thanks to multiple inheritance

↳ In Python **NON** esistono le interfacce!

# Polymorphism

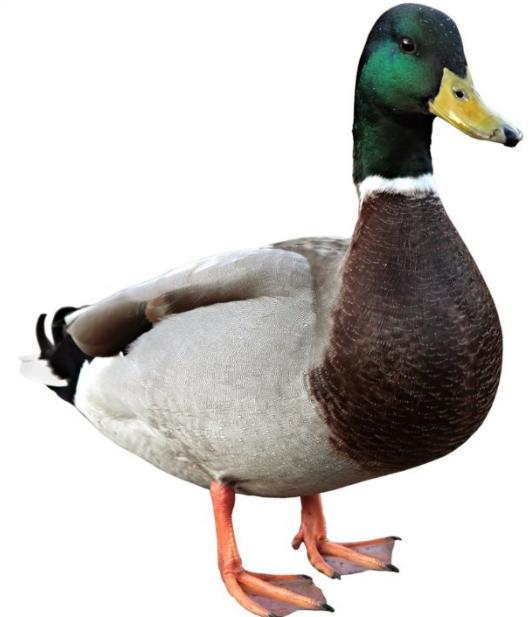
- Polymorphism = calling the same method / function / operation, with different data types
- Java examples:
  - With **sub-classes**: *Il polimorfismo in java veniva fatto con le sottoclassi o con i metodi definiti in overloading* ↗  
L'overloading in Python non può esserci perché non posso avere due metodi nella stessa classe con lo stesso nome ma che ricevono parametri ≠ in input!
  - With **overloaded methods**: public double area(Polygon p) and public double area(Conic c)
- Java selects which method to call based on the signature of the methods and of the inheritance relationships

# Polymorphism in Python

- In Python, method parameters don't have a type specification: cannot check for subclasses or signatures
- Python uses a strategy called “Duck Typing”

“ If it walks like a duck and it quacks like a duck, then it must be a duck ”

Si duck typing viene utilizzato x le **polimorfismo** in python!



# Duck typing

La classe di un oggetto è - importante  
dei metodi che la classe definisce!

- The type or the class of an object is less important than the methods it defines
- When you use duck typing, you do not check types at all. Instead, you check for the presence of a given method or attribute

# Example (1)

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

- In python, i metodi sono più importanti degli attributi che implementa rispetto alla classe a cui appartiene!

The function may be called with totally different classes as parameters

- Sia `source_database` che `source_file` implementano il metodo `read_data()`, pur essendo oggetti diversi!

Io non so che oggetto sia `data_provider` e non mi domando neanche sul tipo di oggetto che sto trattando: l'importante è che l'oggetto ricevuto in input abbia al suo interno il metodo `read_data()`!

What is the allowed type of `data_provider`?

*Duck typing* says: any class that has a `read_data` method.

```
source_database = DatabaseAccess('localhost', 'root', 'root', 'data')  
pretty_print(source_database)  
  
source_file = FileAccess('data.csv')  
pretty_print(source_file)
```

# Example (2)

```
class DatabaseAccess():
    def __init__(self, server, username, password, database):
        self.connection = mysql.connector.connect(server, username, password, database)

    def read_data(self):
        cursor = self.connection.cursor()
        cursor.execute('SELECT * FROM numbers')
        result = cursor.fetchall()
        return result
```

Two unrelated classes, both implementing a `read_data` method, are interchangeable in `pretty_print`.

```
class FileAccess():
    def __init__(self, file_name):
        self.file_name = file_name

    def read_data(self):
        with open(self.file_name, 'r') as f:
            lines = f.readlines()
        result = []
        for line in lines:
            result.append(line.rstrip().split(','))
        return result
```

# Polymorphism

- Inside a polymorphic function, you may check the classes of the received instances. Useful to avoid errors before calling methods that might not exist.
- Do not abuse, it defeats the simplicity of Duck Typing

Con la funzione `isinstance` posso comunque verificare se un oggetto è istanza di una certa classe ~ Non è consigliato utilizzarlo, bisogna fidarsi del duck typing!

## `isinstance(object, classinfo)`

Return `True` if the `object` argument is an instance of the `classinfo` argument, or of a (direct, indirect, or [virtual](#)) subclass thereof. If `object` is not an object of the given type, the function always returns `False`. If `classinfo` is a tuple of type objects (or recursively, other such tuples) or a [Union Type](#) of multiple types, return `True` if `object` is an instance of any of the types. If `classinfo` is not a type or tuple of types and such tuples, a [TypeError](#) exception is raised. [TypeError](#) may not be raised for an invalid type if an earlier check succeeds.

# 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
- La funzione `len()` la posso chiamare su qualunque oggetto che implementa il metodo `__len__()`!*
- ~ Quando se definisco il metodo `__len__()` in una mia classe, potrò chiamare la funzione `len()` su un oggetto di quella classe!*

<https://mypy.readthedocs.io/en/stable/protocols.html#predefined-protocol-reference>

# A Well-Defined class

- To correctly interoperate in the Python world, your class must define
  - An `__init__()` method
  - A set of `self.name` instance attributes initialized in the `__init__()` method
  - A `__repr__()` method for conversion to a (programmer-oriented) string
  - An `__eq__()` method for allowing `==` and `!=` comparisons
  - If required, ordering methods such as `__le__()` for allowing `<` `>` `<=` `>=` 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

Una classe per interagire correttamente con le varie funzioni e classi di Python, dovrà implementare questa serie di protocolli:

- `__init__`: costruttore, per creare oggetti
- `__repr__` / `__str__`: per stampare le info sull'oggetto
- `__eq__` / `__le__`: per confrontare oggetti della stessa classe
- `__hash__`: per utilizzare gli oggetti della classe nei set o come chiavi di un dizionario

# Dataclasses

In ECLIPSE utilizzavamo delle scorciatoie che ci permettevano di creare automaticamente dei metodi "preimpostati", ovvero metodi che andavano definiti con frequenza in tutte le classi. Come visto nella slide precedente, anche in Python abbiamo dei metodi che andranno definiti in tutte le classi per poter essere utilizzate correttamente e al 100% delle possibilità. Anche in Python abbiamo una scorciatoia: le **dataclasses**: tutti i metodi standard possono essere implementati in automatico dichiarando che una determinata classe è una dataclass. Per poterlo fare devo mettere `@dataclass` nella riga di codice precedente alla definizione della classe.

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

Python

```
class RegularCard:  
    def __init__(self, rank, suit):  
        self.rank = rank  
        self.suit = suit
```

...plus boilerplate dunder methods

Quando vado a decorare una classe col decoratore `@dataclass`, sto generando automaticamente i metodi `__init__`, `__str__`, `__repr__` ecc. sulla base degli attributi che le dichiaro.

Gli attributi vanno definiti in un modo strano,  
`nome_attributo : tipo_attributo`

Python

```
from dataclasses import dataclass  
  
@dataclass  
class DataClassCard:  
    rank: str  
    suit: str
```

`dataclass` va importato dalla libreria `dataclasses`!

- `@dataclass` decorator
- List of attributes
- Expected types of attributes, after semicolon

Se la dataclass mi definisce `__eq__(self)` e io lo voglio ridefinire in modo `!=`, posso farlo e quando chiamerò il metodo `__eq__(self)`, verrà considerata la mia implementazione e non quella del `dataclass`!

<https://docs.python.org/3/library/dataclasses.html>

<https://realpython.com/python-data-classes/>

Il tipo-attributo chiaramente verrà ignorato da python ma servirà all'IDE!



# License

- These slides are distributed under a Creative Commons license “**Attribution-NonCommercial-ShareAlike 4.0 International (CC BY-NC-SA 4.0)**”
- **You are free to:**
  - **Share** — copy and redistribute the material in any medium or format
  - **Adapt** — remix, transform, and build upon the material
  - The licensor cannot revoke these freedoms as long as you follow the license terms.
- **Under the following terms:**
  - **Attribution** — You must give appropriate credit, provide a link to the license, and indicate if changes were made. You may do so in any reasonable manner, but not in any way that suggests the licensor endorses you or your use.
  - **NonCommercial** — You may not use the material for commercial purposes.
  - **ShareAlike** — If you remix, transform, or build upon the material, you must distribute your contributions under the same license as the original.
  - **No additional restrictions** — You may not apply legal terms or technological measures that legally restrict others from doing anything the license permits.
- <https://creativecommons.org/licenses/by-nc-sa/4.0/>