Programming Lab

Parte 4

Gli oggetti in Python

Laura Nenzi

Many slides by Stefano Russo

Programmazione ad oggetti

E’ un paradigma di programmazione. Cambia molto.

Permette di ragionare in termini di entità e non solo di funzioni.

Gli ***oggetti*** sono definiti con le ***classi***

Una classe è uno schema che permette di descrivere un’entità con le caratteristiche desiderate.

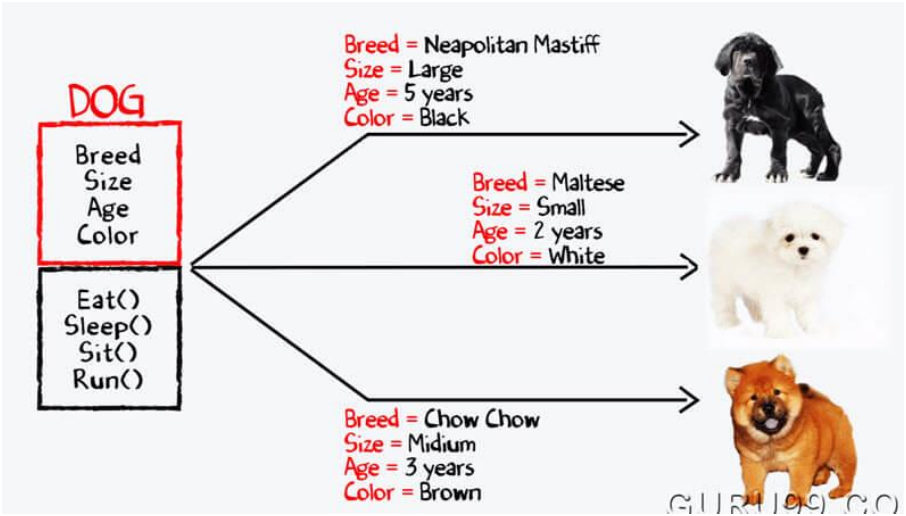
Programmazione ad oggetti

Negli oggetti/classi:

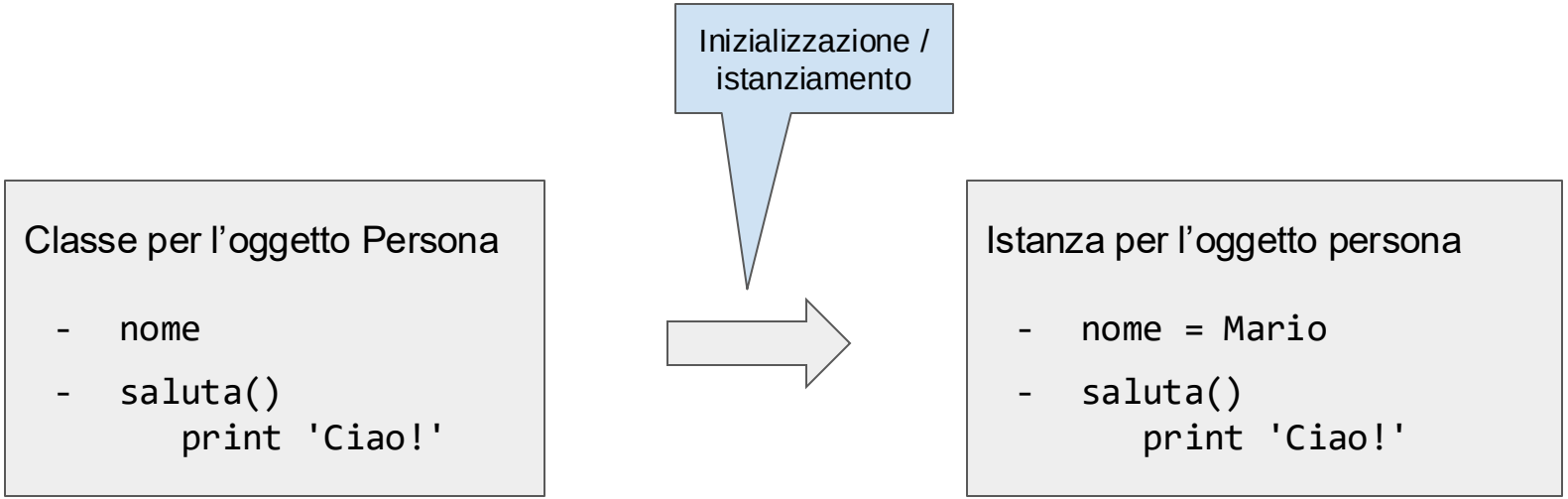
- le funzioni negli si chiamano ***metodi***
- le variabili negli si chiamano ***attributi***

Una volta inizializzati diventano ***istanze***
→ si parla infatti di *istanziare* una classe

Programmazione ad oggetti



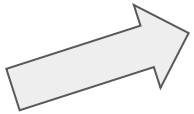
Programmazione ad oggetti

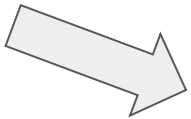


Programmazione ad oggetti

Classe per l'oggetto Persona

- `nome`
- `saluta()`
  `print 'Ciao!'`





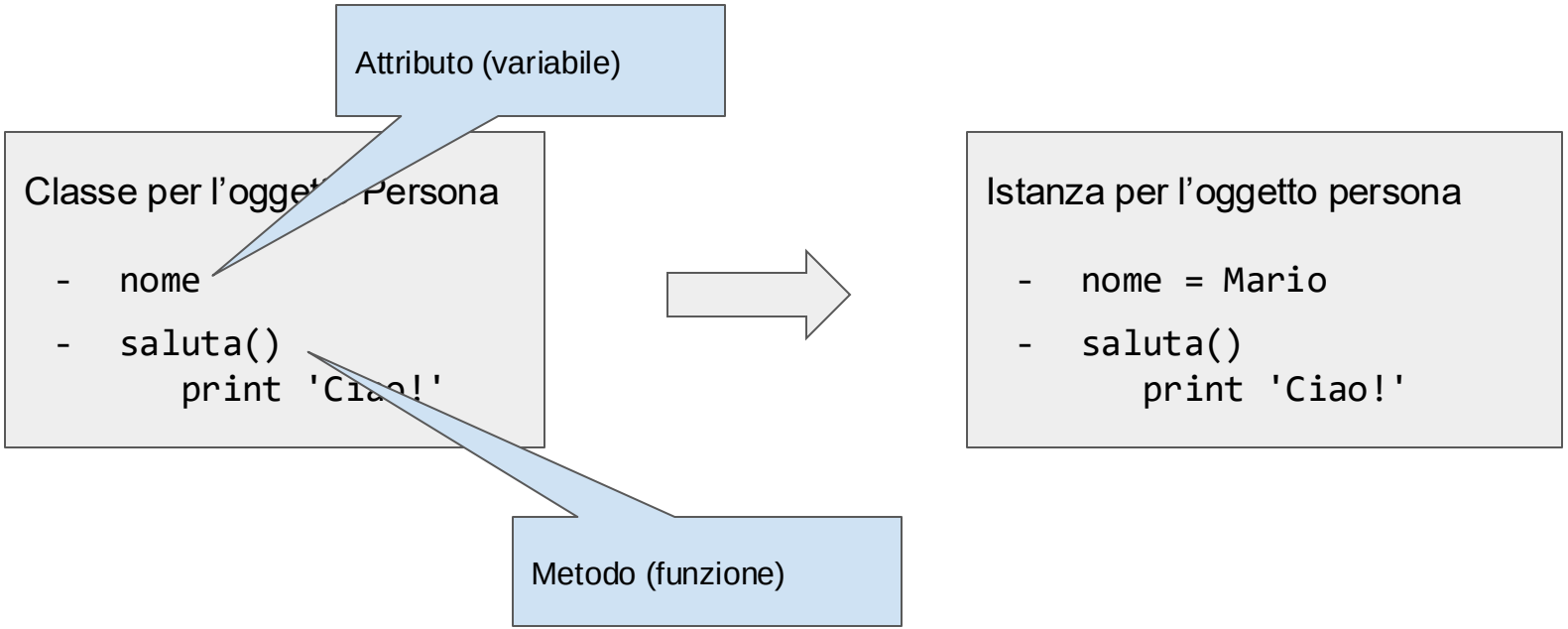
Istanza per l'oggetto persona

- `nome = Mario`
- `saluta()`
  `print 'Ciao!'`

Istanza per l'oggetto persona

- `nome = Lucia`
- `saluta()`
  `print 'Ciao!'`

Programmazione ad oggetti



Programmazione ad oggetti

Gli oggetti si usano principalmente perchè:

- Permettono di rappresentare bene delle gerarchie (e sfruttare le caratteristiche in comune)

- Una volta istanziati, permettono di mantenere facilmente lo *stato* (senza diventare matti con strutture dati di appoggio)

Convenzioni

In Python c’è una convenzione di stile ben precisa:

- notazione **snake_case** (caratteri minuscoli e underscore) per i nomi dei **metodi,** delle **variabili** e delle **istanze** degli oggetti
- notazione **CamelCase** per il nome delle **classi**

Inoltre, doppi underscore prima e dopo il nome di un metodo indicano un metodo ad uso esclusivamente interno (esempio: `__str__`, oppure `__doc__`)

Gli apici valgono sia singoli che doppi, ma conviene usarli singoli per il codice, doppi nelle stringhe, `e.g. mystring = 'Il mio nome è "Mario" e sono una persona'`

In Python tutto è un oggetto

```
>>> my_string_2 = 'corso di laboratorio di programmazione'
>>> dir(my_string_2)
['__add__', '__class__', '__contains__', '__delattr__', '__dir__', '__doc__', '__eq__', '__for
mat__', '__ge__', '__getattribute__', '__getitem__', '__getnewargs__', '__gt__', '__hash__', '
__init__', '__init_subclass__', '__iter__', '__le__', '__len__', '__lt__', '__mod__', '__mul__
', '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__', '__rmod__', '__rmul__', '__
setattr__', '__sizeof__', '__str__', '__subclasshook__', 'capitalize', 'casefold', 'center', '
count', 'encode', 'endswith', 'expandtabs', 'find', 'format', 'format_map', 'index', 'isalnum'
, 'isalpha', 'isascii', 'isdecimal', 'isdigit', 'isidentifier', 'islower', 'isnumeric', 'ispri
ntable', 'isspace', 'istitle', 'isupper', 'join', 'ljust', 'lower', 'lstrip', 'maketrans', 'pa
rtition', 'replace', 'rfind', 'rindex', 'rjust', 'rpartition', 'rsplit', 'rstrip', 'split', 's
plitlines', 'startswith', 'strip', 'swapcase', 'title', 'translate', 'upper', 'zfill']
>>> my_string_2.title()
'Corso Di Laboratorio Di Programmazione'
```

In Python tutto è un oggetto

examples.py

```
my_string = 'a,b,c'
print(my_string)
print(my_string.split(','))
print(my_string)
```

```
> python examples.py
a,b,c
['a', 'b', 'c']
a,b,c
```

examples.py

```
my_list = [1,2,3,4]
print(my_list)
print(my_list.reverse())
print(my_list)
```

```
> python examples.py
[1, 2, 3, 4]
None
[4, 3, 2, 1]
```

Oggetti con operazioni in-place e non

examples.py

```
my_string = 'a,b,c'
print(my_string)
print(my_string.split(','))
print(my_string)
```

Questa è un'operazione (funzione, metodo) che quando viene eseguita torna il risultato

```
> python examples.py
a,b,c
['a', 'b', 'c']
a,b,c
```

examples.py

```
my_list = [1,2,3,4]
print(my_list)
print(my_list.reverse())
print(my_list)
```

Questa è un'operazione (funzione, metodo) **in-place** che quando viene eseguita modifica l'oggetto, non torna niente!

```
> python examples.py
[1, 2, 3, 4]
None
[4, 3, 2, 1]
```

Definire oggetti

objects.py

```python
class Person():
    pass

person = Person()
print(person)
```

```
> python objects.py
<__main__.Person object at 0x7ff378a93fa0>
>
```

Definire oggetti

objects.py

```
class Person():
    pass

person = Person()
print(person)
```

"pass" è l'istruzione nulla, serve per avere un blocco vuoto

Questa operazione *istanzia* una classe di oggetto.

Ovvero, da una generica definizione di un oggetto ne creo uno specifico.

```
> python objects.py
<__main__.Person object at 0x7ff378a93fa0>
>
```

Definire oggetti

objects.py

```python
class Person():

    def __init__(self, name, surname):

        # Set name and surname
        self.name    = name
        self.surname = surname

person = Person('Mario', 'Rossi')
print(person)
print(person.name)
print(person.surname)
```

```
> python objects.py
<__main__.Person object at 0x7f8a75ac0fa0>
Mario
Rossi
>
```

Definire oggetti

la funzione _init_ (costruttore) è quella che è responsabile di inizializzare l'oggetto, i.e. istanzia gli attributi.
Se non c'è viene usata quella di default che non fa nulla.

objects.py

```
class Person():

    def __init__(self, name, surname):

        # Set name and surname
        self.name    = na
        self.surname = surn

person = Person('Mario', 'Rossi
print(person)
print(person.name)
print(person.surname)
```

```
> python objects.py
<__main__.Person object at 0x7f8a75ac0fa0>
Mario
Rossi
>
```

"self" vuol dire "me stesso", "me *istanza* di classe". E' obbligatorio come parametro in tutti i metodi degli oggetti (salvo casi particolari)

È prassi che i parametri di __init__ abbiano gli stessi nomi degli attributi. Se metto dei valori di default diventano opzionali.
Buona prassi è inizializzare tutti gli attributi di un oggetto.

Definire oggetti

objects.py

```python
class Person():

    def __init__(self, name, surname):

        # Set name and surname
        self.name    = name
        self.surname = surname

    def __str__(self):
        return 'Person "{} {}"'.format(self.name, self.surname)

person = Person('Mario', 'Rossi')
print(person)
```

```
> python objects.py
Person "Mario Rossi"
>
```

Definire oggetti

objects.py

```
class Person():

    def __init__(self, name, surname):

        # Set name and surname
        self.name    = name
        self.surname = surname

    def __str__(self):
        return 'Person "{} {}"'.format(self.name, self.surname)

person = Person('Mario', 'Rossi')
print(person)
```

Funzione ad uso interno che vado a sovrascrivere. E' responsabile della rappresentazione in formato stringa dell'oggetto.

```
> python objects.py
Person "Mario Rossi"
>
```

objects.py

```python
# Import the random module
import random

class Person():

    def __init__(self, name, surname):

        # Set name and surname
        self.name      = name
        self.surname   = surname

    def __str__(self):
        return 'Person "{} {}""'.format(self.name, self.surname)

    def say_hi(self):

        # Generate a random number between 0, 1 and 2.
        random_number = random.randint(0,2)

        # Choose a random greeting
        if random_number == 0:
            print('Hello, I am {} {}.'.format(self.name, self.surname))
        elif random_number == 1:
            print('Hi, I am {}!'.format(self.name))
        elif random_number == 2:
            print('Yo bro! {} here!'.format(self.name))

person = Person('Mario', 'Rossi')
person.say_hi()
```

```
> python objects.py
Hello, I am Mario Rossi.
>
```

```
> python objects.py
Hi, I am Mario!
>
```

```
> python objects.py
Yo bro! Mario here!
>
```

objects.py

```python
# Import the random module
import random

class Person():

    def __init__(self, name, surname):

        # Set name and surname
        self.name      = name
        self.surname   = surn

    def __str__(self):
        return 'Person "{} {}"'.format(self.name, self.surname)

    def say_hi(self):

        # Generate a random number between 0, 1 and 2.
        random_number = random.randint(0,2)

        # Choose a random greeting
        if random_number == 0:
            print('Hello, I am {} {}.'.format(self.name, self.surname))
        elif random_number == 1:
            print('Hi, I am {}!'.format(self.name))
        elif random_number == 2:
            print('Yo bro! {} here!'.format(self.name))

person = Person('Mario', 'Rossi')
person.say_hi()
```

Funzione (metodo) dell'oggetto. Anche chiamata interfaccia. Sono le funzioni che verranno testate per l'esame!

```
> python objects.py
Hello, I am Mario Rossi.
>
```

```
> python objects.py
Hi, I am Mario!
>
```

```
> python objects.py
Yo bro! Mario here!
>
```

Gli oggetti sono mutabili

- Posso anche assegnari nuovi valori agli attributi,

```
person = Person("Mario", "Rossi")
print(person)
person.surname = "Verdi"
print(person)
```

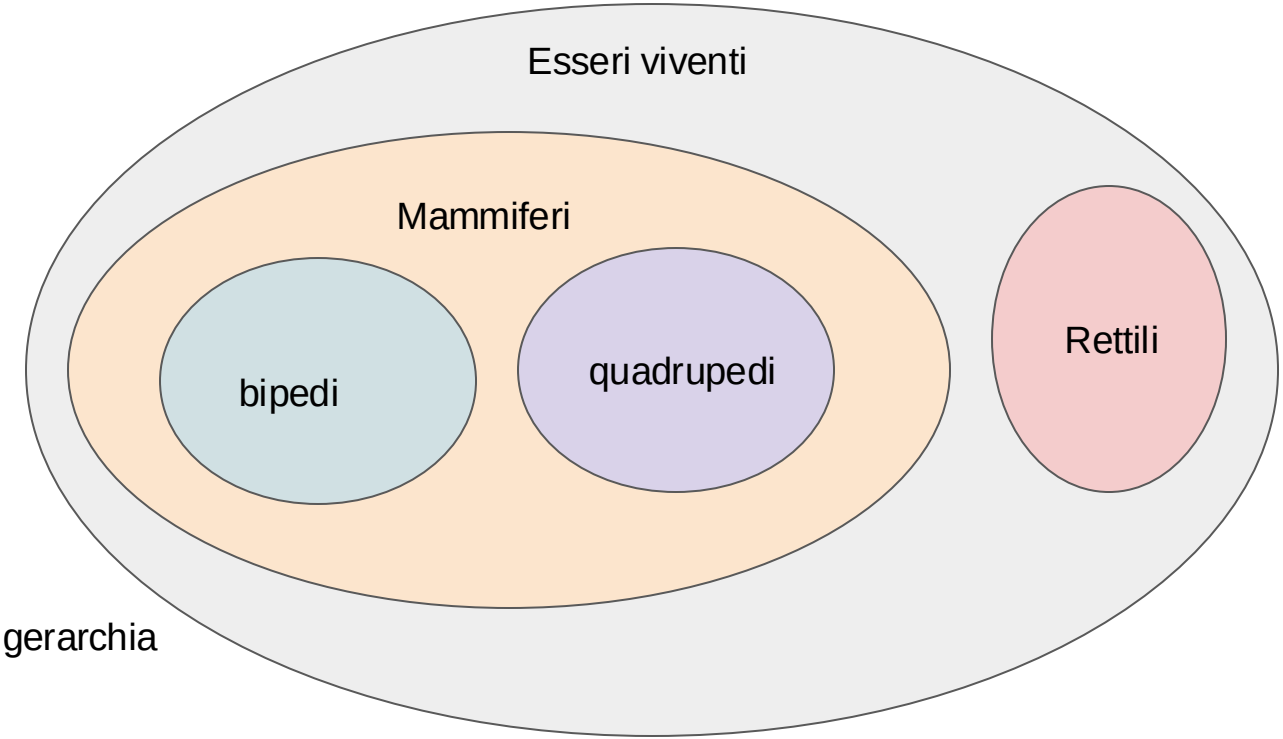
```
(.conda) (base) laurane
Person "Mario Rossi"
Person "Mario Verdi"
```

- Se la classe base ha un costruttore __init__(), la classe figlia può averne uno diverso con solo parte degli attributi. In quel caso posso specificare l'attributo della classe base con super().__init__()

Estendere oggetti: l'ereditarietà

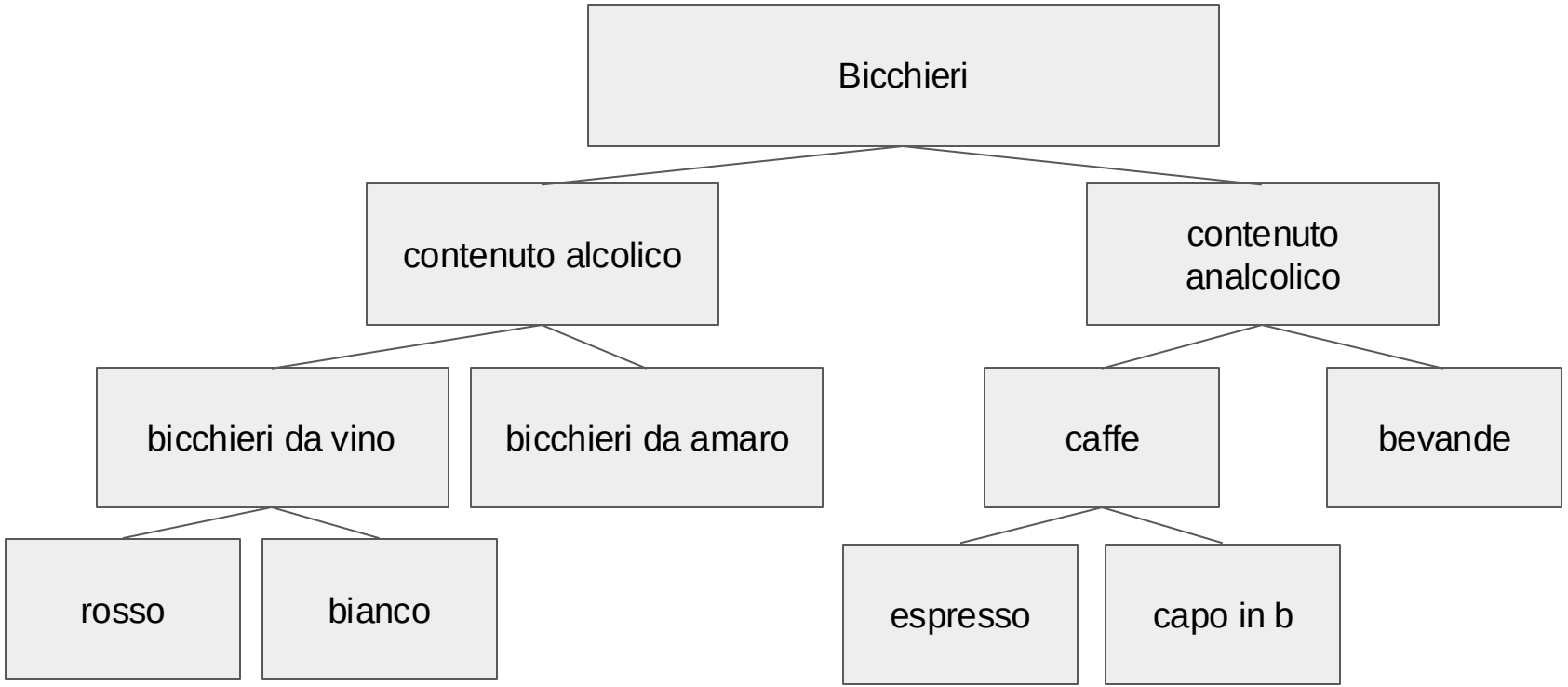
- Gli oggetti sono anche molto comodi per rappresentare gerarchie
- Il concetto di ereditarietà permette, a partire da una classe genitore, di creare classi figlie, cioè di definire nuove classi come versione modificata di una classe già esistente.
- Possiamo interpretare le gerarchie tra classi in termini di insiemi e relazioni di inclusione

Estendere oggetti : l'ereditarietà

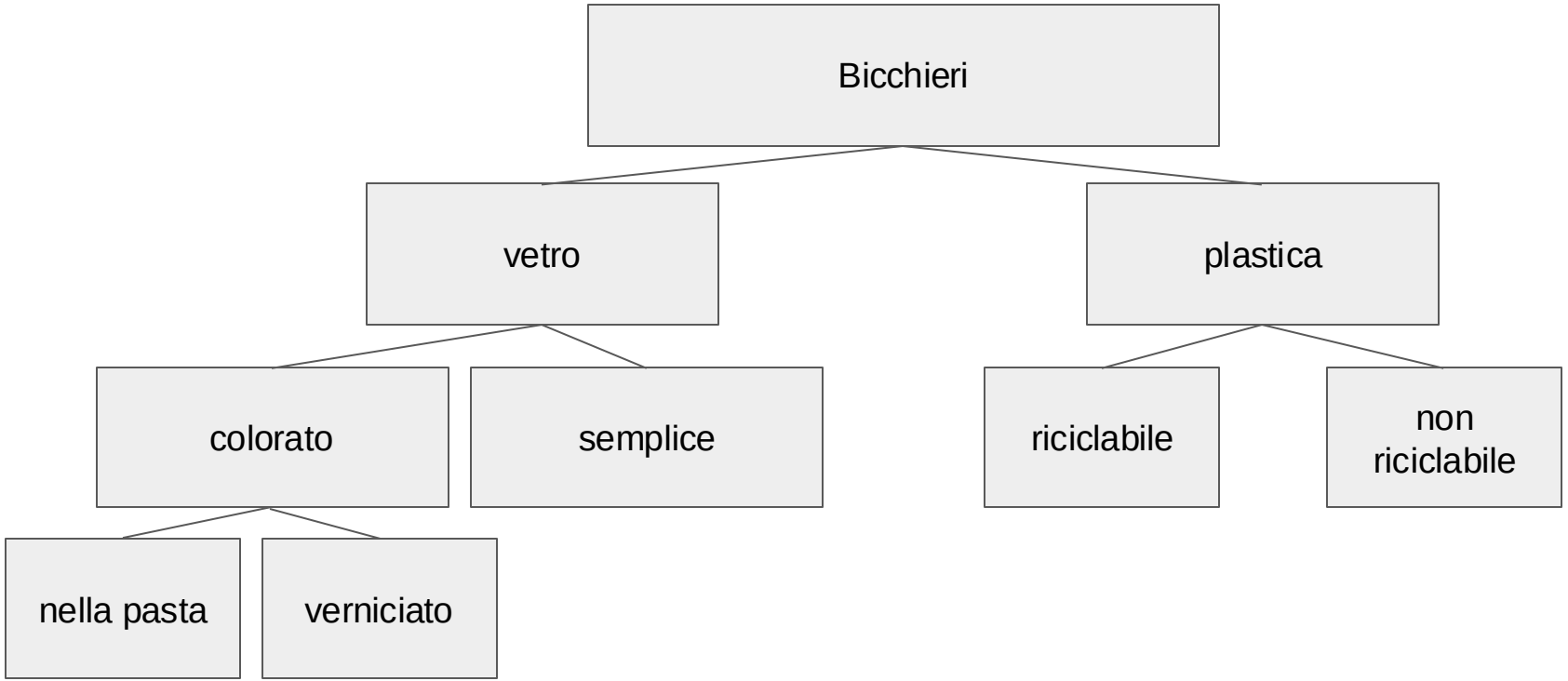


Esempio di gerarchia

Estendere oggetti



Estendere oggetti



Estendere oggetti

- L’operazione di estensione di una classe prende anche il nome di **specializzazione**, nel senso che la si rende più “specifica” per una determinata entità che si vuole rappresentare

- Le classi figlie:
  - ereditano tutti gli attributi e metodi della classe genitore
  - possono sovrascrivere i metodi ereditati per personalizzarli e possono aggiungerne di nuovi per estenderne le funzionalità.

- Le modifiche effettuate ad una classe figlia non impattano la classe genitore.

Estendere oggetti

objects.py

```
class Person():
    ...

class Student(Person):

    def __str__(self):
        return 'Student "{} {}"'.format(self.name, self.surname)


class Professor(Person):

    def __str__(self):
        return 'Prof. "{} {}"'.format(self.name, self.surname)

    def say_hi(self):
        print('Hello, I am professor {} {}.'.format(self.name, self.surname))
```

Estendere oggetti

objects.py

```
class Person():
    ...

class Student(Person):

    def __str__(self):
        return 'Student "{} {}"'.format(self.name, self.surname)


class Professor(Person):

    def __str__(self):
        return 'Prof. "{} {}"'.format(self.name, self.surname)

    def say_hi(self):
        print('Hello, I am professo{} {}.'.format(self.name, self.surname))
```

Estendo l'oggetto Persona declinandolo in Studente e Professore. Tutti i metodi che possedeva l'oggetto Persona sono automaticamente ereditati dagli oggetti Persona e Professore. Posso sovrascriverli o aggiungerne altri.

Sovrascrivo la rappresentazione in stringa dell'oggetto Persona per includere il titolo.

Sovrascrivo il metodo che saluta dell'oggetto Persona per avere un saluto di più consono ad un professore.

Estendere oggetti

objects.py

```
class Person():
    ...

class Student(Person):

    def __str__(self):
        return 'Student "{} {}"'.format(self.name, self.surname)


class Professor(Person):

    def __str__(self):
        return 'Prof. "{} {}"'.format(self.name, self.surname)

    def say_hi(self):
        print('Hello, I am professor {} {}.'.format(self.name, self.surname))

    def original_say_hi(self):
        super().say_hi()
```

Estendere oggetti

objects.py

```
class Person():
    ...

class Student(Person):

    def __str__(self):
        return 'Student "{} {}"'.format(self.name, self.surname)


class Professor(Person):

    def __str__(self):
        return 'Prof. "{} {}"'.format(self.name, self.surname)

    def say_hi(self):
        print('Hello, I am professor {} {}.'.format(self.name, self.surname))

    def original_say_hi(self):
        super().say_hi()
```

Con il “super” accedo alla funzione dell’oggetto padre, anche se l’ho sovrascritta

Estendere oggetti

objects.py

```
class Person():
    ...

class Student(Person):

    def __str__(self):
        return 'Student "{} {}"'.format(self.name, self.surname)


class Professor(Person):

    def __str__(self):
        return 'Prof. "{} {}"'.format(self.name, self.surname)

    def say_hi(self):
        print('Hello, I am professor {} {}.'.format(self.name, self.surname))

    def original_say_hi(self):
        super().say_hi()
```

Esempio

```
print('----------------------------')

person = Person('Mario', 'Rossi')

print(person)
person.say_hi()

print('----------------------------')

prof = Professor('Pippo', 'Baudo')

print(prof)
prof.say_hi()
prof.original_say_hi()

print('----------------------------')
```

```
> python objects.py
----------------------------
Person "Mario Rossi""
Yo bro! Mario here!
----------------------------
Prof. "Pippo Baudo"
Hello, I am professor Pippo Baudo.
Yo bro! Pippo here!
----------------------------
```

Stile di codice

- Usa un'**indentazione di 4 spazi** e non usare tabulazioni. Le tabulazioni introducono confusione e sarebbe meglio evitarle.
- Suddividi le righe in modo che non superino **79 caratteri**. Questo aiuta gli utenti con display piccoli e rende possibile visualizzare più file di codice affiancati su schermi più grandi.
- Usa righe vuote per separare funzioni e classi, oltre che per separare blocchi di codice più grandi all'interno delle funzioni.
- Quando possibile, inserisci i commenti su una loro riga separata.
- Usa le **docstring**.
- Usa spazi intorno agli operatori e dopo le virgole, ma **non direttamente all'interno delle parentesi**: ad esempio a = f(1, 2) + g(3, 4).

Stile di codice

- Dai un nome alle tue classi e funzioni in modo coerente, **CamelCase** per le classi, **snake_case** per funzioni e variabili.
- Usa sempre **self** come nome per il primo argomento di un metodo
- Non utilizzare codifiche complesse se il tuo codice è destinato a essere utilizzato in ambienti internazionali. Il semplice **ASCII** funziona meglio in ogni caso.
- Tutte le convenzioni e molto altro lo trovare nei PEP (Python Enhancement Proposal). Servono come documentazione tecnica per spiegare le motivazioni, i dettagli e gli effetti delle modifiche proposte.
- Il PEP 8 - Style Guide for Python Code: https://peps.python.org/pep-0008/

Esercizio 1

Create un oggetto **CSVFile** che rappresenti un file CSV, e che:

1) venga inizializzato sul nome del file csv, e
2) abbia un attributo “name” che ne contenga il nome
3) abbia un metodo “get_data()” che torni i dati dal file CSV come lista di liste, ad es: `[ ['01-01-2012', '266.0'], ['01-02-2012', '145.9'], ... ]`

Provatelo sul file “shampoo_sales.csv”.

Poi, scaricate il vostro script Python e testatelo su autograding

p.s. il massimo punteggio è 9/10 con quello che abbiamo visto fino ad oggi a lezione.

Esercizio 2: **Classe Veicolo**

Scrivete una classe denominata **Veicolo** che disponga di questi attributi dati:

- **modello** (per il modello del veicolo);
- **marca** (per la marca del veicolo);
- **anno** (per l'anno del veicolo).
- **speed** (per la velocità del veicolo)

E di questi metodi:

- __init__ che accetti come argomenti l'anno, il modello, e la marca. Il metodo dovrebbe inoltre assegnare **0** all'attributo dati speed.
- __**str**__ che restituisce una stringa con i dettagli del veicolo (marca, modello, anno e velocità)
- **accellerare** che aggiunge **5** all'attributo dati speed ogni volta che viene chiamato.
- **frenare** che sottrae **5** dall'attributo dati speed ogni volta che viene chiamato.
- **get_speed** che restituisce la velocità corrente.

Esercizio 3

- Crea una sottoclasse auto che ha in aggiunta l'attributo **numero_porte** e cambia il metodo _str__ di conseguenza
- Crea una sottoclasse moto che ha in aggiunta l'attributo **tipo** (ad esempio, "Sportiva" o "Touring") e cambia il metodo _str__ di conseguenza