Pattern Recognition
and Applications Lab

La Programmazione ad Oggetti in Python

Docente: Ambra Demontis

Anno Accademico: 2020 - 2021

University of Cagliari, Italy

Department of Electrical and Electronic Engineering

Corso di Laurea in Ingegneria Elettrica, Elettronica e Informatica

La Programmazione ad Oggetti in Python

In queste slide vedremo come si implementano le relazioni in Python, in particolare:

- La relazione di composizione
- La relazione di aggregazione
- La relazione di ereditarietà

La relazione di Composizione

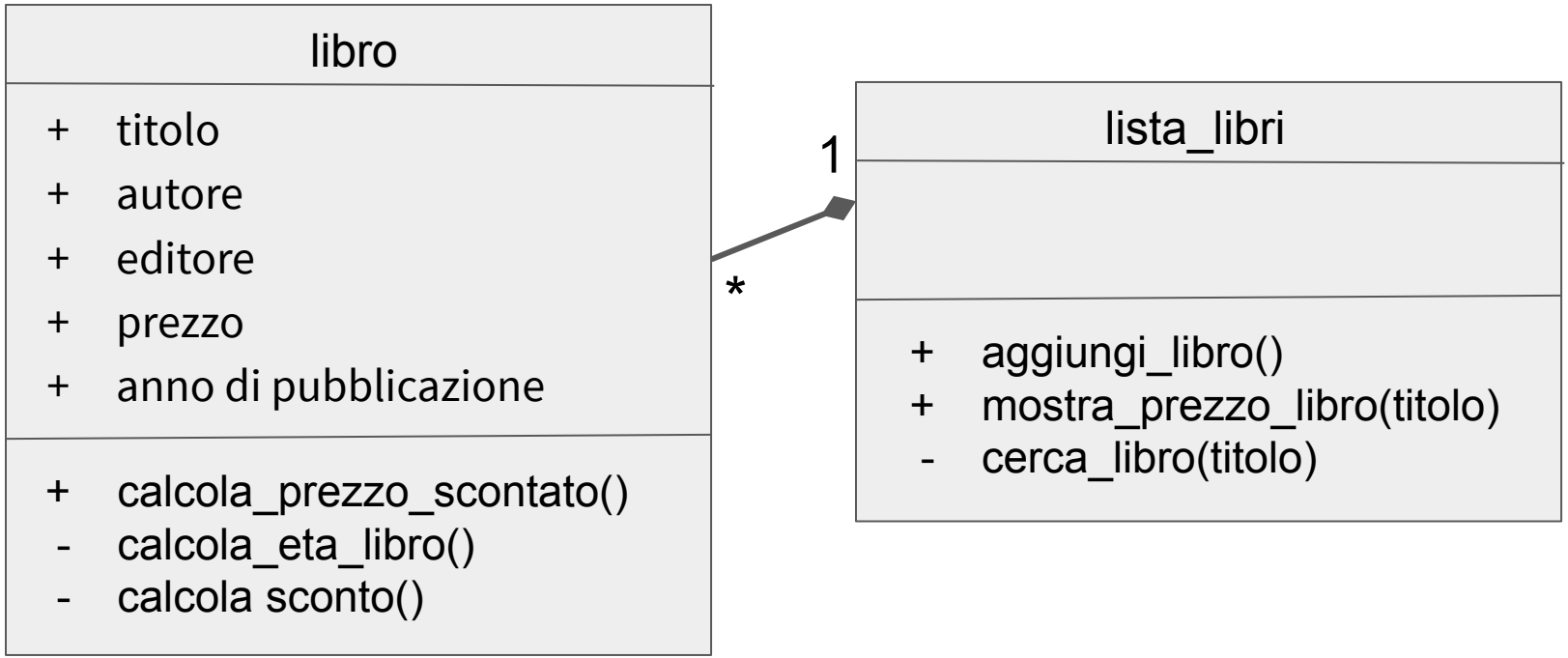
Nella lezione precedente abbiamo visto come definire una classe. In particolare abbiamo creato la classe CLibro.

Supponiamo ora di voler memorizzare più libri e quindi di creare una classe per gestire una lista dei libri, composta da libri appartenenti alla classe CLibro.

Lo scopo della classe che gestisce la lista dei libri è di:
-memorizzare un nuovo libro
-mostrare il prezzo finale (scontato) di un libro

La relazione di Composizione

Il diagramma delle classi del prototipo è il seguente:



Le Strutture Dati di Python e gli Oggetti

Per poter creare questa classe abbiamo bisogno di memorizzare diversi oggetti di tipo libro.

Le strutture dati tupla, lista e dizionario possono memorizzare oggetti.

In questo caso, poichè vogliamo memorizzare un insieme di dati omogeneo (sono tutti oggetti appartenenti alla classe CLibro), utilizzeremo una lista.

Creiamo la Classe CListaLibri

La prima funzione che vogliamo implementare è la funzione “aggiungi_libro”.
Come spiegato nella slide precedente è possibile memorizzare gli oggetti in una lista.

Quando dobbiamo creare questa lista?
In teoria potremmo crearla la prima volta che aggiungiamo un libro.
Tuttavia **è consigliabile inizializzare tutti gli attributi, pubblici e privati nel metodo __init__ dell’oggetto.**

In questo modo un programmatore che vuole modificare la classe creata guardando il costruttore è a conoscenza di tutti gli attributi definiti da quella classe.

Le Strutture Dati di Python e gli Oggetti

Che valore gli si deve assegnare quando essi ricevono un valore successivamente?

Nel caso in cui si tratti di:

- Una struttura dati tra: tupla, lista o dizionario, gli si assegna una struttura dati di quel tipo vuota.
- Un altro tipo di dato: gli si assegna il tipo di dato None (nessun valore).

L’ Operatore Is

Per controllare se una variabile è None si utilizza l’operatore is.
La sintassi è la seguente:
<nome_variabile> is None

Lo stesso operatore si utilizza anche per controllare se una variabile è True o False.

Creiamo la Classe CListaLibri

Abbiamo detto che avremo bisogno di una lista per memorizzare gli oggetti di classe CLibro e che è meglio crearla nel metodo __init__ dell'oggetto.

```
class CListaLibri:

  def __init__(self):
    self._lista = []
```

Creiamo la Classe CListaLibri

Creiamo la funzione pubblica *aggiungi_libro* che deve permettere di memorizzare un libro nella lista.

Per avere funzioni più compatte possiamo far si che il nuovo libro venga creato utilizzando un'apposita funzione privata.

```
def aggiungi_libro(self):
  # acquisisce i dati di un nuovo libro e crea un oggetto di tipo CLibro
  oggetto_libro = self._crea_nuovo_oggetto_libro()
  self._lista = self._lista + [oggetto_libro]
```

Creiamo la Classe CListaLibri

Implementiamo la funzione privata *_crea_nuovo_oggetto_libro* che si occupa di creare un oggetto di tipo libro ed acquisire i valori dei suoi attributi.

```
def _crea_nuovo_oggetto_libro(self):
  # acquisisci i dati
  titolo = input("inserisci il titolo del libro ")
  autore = input("inserisci l'autore del libro ")
  editore = input("inserisci l'editore del libro ")
  prezzo = float(input("inserisci il prezzo del libro "))
  anno_pubblicazione = int(
     input("inserisci l'anno di pubblicazione del libro "))
```

```
  # crea un oggetto appartenente alla classe CLibro
  oggetto_libro = CLibro(titolo, autore,
                         editore, prezzo,
                         anno_pubblicazione)
  return oggetto_libro
```

Creiamo la Classe CListaLibri

Implementiamo la funzione privata *_cerca_libro* che, dato il titolo di un libro, cerca il libro nella lista e ci restituisce l'oggetto corrispondente.

Un modo per implementarla è il seguente:

```
def _cerca_libro(self, titolo):
  libro_trovato = False
  for libro in self._lista:
     if libro.titolo == titolo:
        libro_cercato = libro
        libro_trovato = True
  if libro_trovato:
     return libro_cercato
  else:
     print("Libro non trovato!")
```

Creiamo la Classe CListaLibri

Questa stessa funzione può essere scritta in modo più conciso sfruttando il fatto che, come nel linguaggio C, quando l'istruzione return viene chiamata, la funzione termina.

```
def _cerca_libro(self, titolo):
  for libro in self._lista:
    if libro.titolo == titolo:
      return libro
  print("Libro non trovato!")
```

Creiamo la Classe CListaLibri

Infine, dobbiamo implementare la funzione *mostra_prezzo_libro* che, dato il titolo di un libro e l'anno corrente, cerca il libro e poi calcola e mostra il prezzo finale del libro utilizzando la funzione *calcola_prezzo_scontato* della classe Clibro.

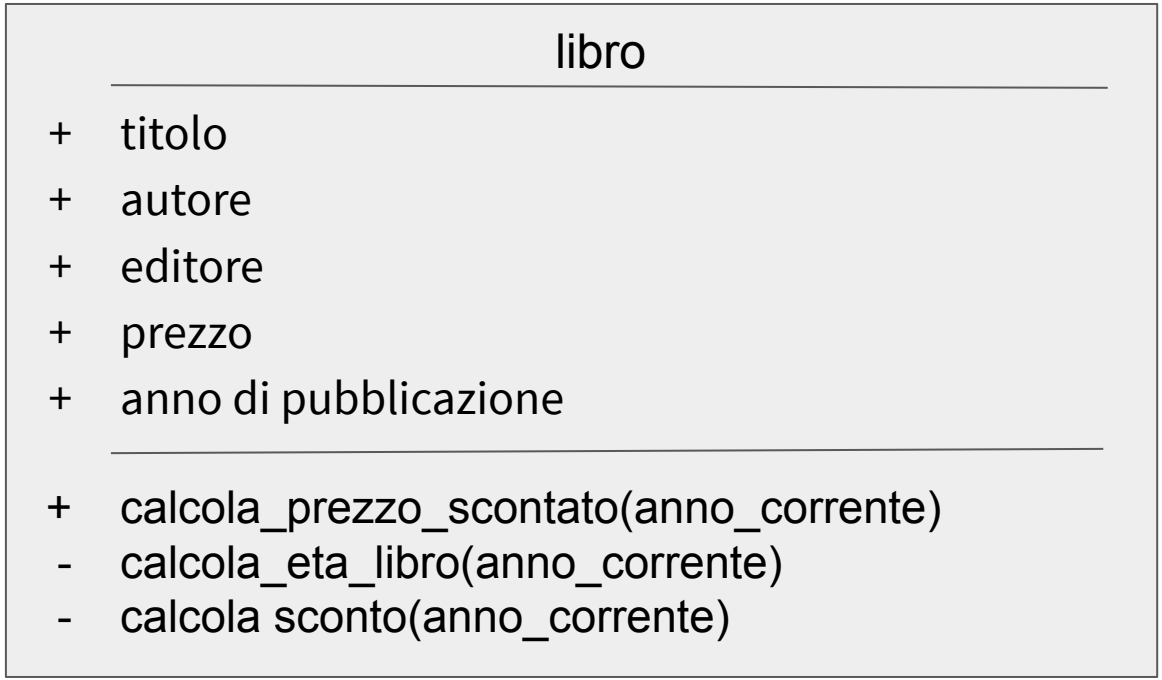
```
def mostra_prezzo_libro(self, titolo, anno_corrente):
  oggetto_libro = self._cerca_libro(titolo)
  # (quando l'istruzione return non viene chiamata, la funzione restituisce None)
  if oggetto_libro is not None:
    print(oggetto_libro.calcola_prezzo_scontato(anno_corrente))
```

Creiamo un’istanza della classe CListaLibri

Proviamo ora ad utilizzare la classe CListalibri aggiungendo due libri e cercando il prezzo scontato di uno dei due libri aggiunti.

```
lista_libri = CListaLibri()
lista_libri.aggiungi_libro()
lista_libri.aggiungi_libro()
lista_libri.mostra_prezzo_libro('libro1', 2021)
```

Il Diagramma di Classe della classe CLibro implementata



Il Diagramma di Classe della classe ClistaLibri implementata

Diagrammi di Classe delle Implementazioni e Prototipi

Generalmente:

- I diagrammi delle classi creati come prototipo servono per avere uno schema di massima di ciò che il programmatore dovrà implementare (sia per ridurre il tempo che l'analista deve impiegare per realizzare lo schema e per ottenere un diagramma più compatto).

- Il programmatore può modificarli per necessità implementative aggiungendo funzioni private per ottenere un codice più compatto, attributi privati o parametri nelle funzioni.

Diagrammi di Classe delle Implementazioni e Prototipi

A volte però, il programmatore può essere vincolato a rispettare rigorosamente il prototipo o parte di esso.

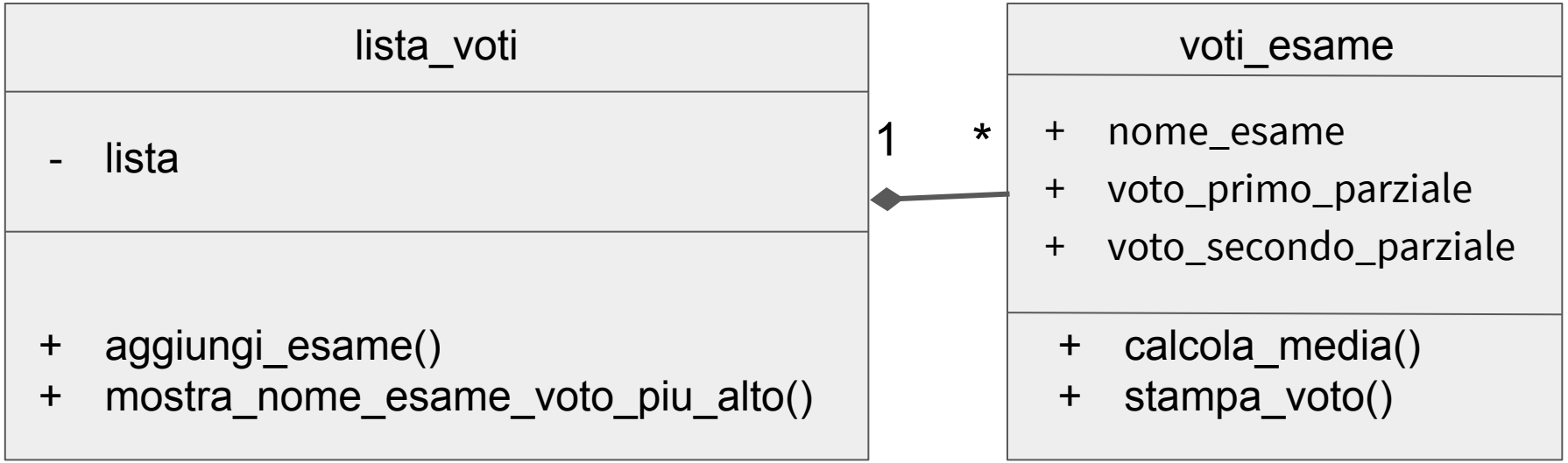
Il caso più comune è quello nel quale il programmatore è obbligato ad utilizzare l'interfaccia specificata per alcuni metodi, in modo che abbiano un' interfaccia comune a quelli di altri oggetti.

Esercizio sulla Composizione in Python

Create la classe *lista_voti*, che deve permettere di:

- Memorizzare diversi oggetti di tipo *CEsame*.
- Stampare il nome dell'esame nel quale è stato ottenuto il voto finale più alto. Nel caso in cui il voto più alto sia lo stesso per più esami, dovrà essere mostrato il nome tutti tutti gli esami nei quali è stato conseguito quel voto.

Esercizio sulla Composizione in Python



Esercizio sulla Composizione in Python

La classe esame sarà molto simile a quella vista precedentemente, l'unico cambiamento sarà il fatto che questa volta la funzione che stampa il voto finale non si chiamerà più _calcola_media ma calcola_media.

Questo metodo infatti ora, come indicato nel diagramma UML deve essere pubblico, in quanto deve essere utilizzata da un'altro oggetto.

Esercizio sulla Composizione in Python

```
class CEsame:

  def __init__(self, nome_esame, voto_primo_parziale, voto_secondo_parziale):
    self.nome_esame = nome_esame
    self.voto_primo_parziale = voto_primo_parziale
    self.voto_secondo_parziale = voto_secondo_parziale

  def calcola_media(self):
    media = (self.voto_primo_parziale + self.voto_secondo_parziale)/2
    return media

  def stampa_voto_finale(self):
    voto_finale = self.calcola_media()
    print("Il voto finale per l'esame ", self.nome_esame, " è ", voto_finale)
```

Esercizio sulla Composizione in Python

```
class CListaVoti:

  def __init__(self):
    self._lista = []

  def aggiungi_esame(self):
    # acquisisce i dati di un nuovo esame e crea un oggetto di tipo CEsame
    oggetto_esame = self._crea_nuovo_oggetto_esame()
    self._lista = self._lista + [oggetto_esame]

  … continua
```

Esercizio sulla Composizione in Python

```
def _crea_nuovo_oggetto_esame(self):
    # acquisisci i dati
    nome_esame = input("inserisci il nome dell'esame ")
    voto_primo_parziale = int(input("inserisci il voto conseguito nel primo parziale "))
    voto_secondo_parziale = int(input("inserisci il voto conseguito nel primo parziale "))

    # crea un oggetto appartenente alla classe CEsame
    oggetto_esame = CEsame(nome_esame, voto_primo_parziale, voto_secondo_parziale)

    return oggetto_esame

    … continua
```

Esercizio sulla Composizione in Python

```
def mostra_nome_esame_voto_piu_alto(self):
    # cerca il voto più alto
    max_voto = 0
    for esame in self._lista:
        voto_finale = esame.calcola_media()
        if voto_finale > max_voto:
            max_voto = voto_finale
```

```
# stampa i nomi degli esami nei quali è
# stato conseguito il voto più alto
    for esame in self._lista:
        voto_finale = esame.calcola_media()
        if voto_finale == max_voto:
            print(esame.nome_esame)
```

Esercizio sulla Composizione in Python

Proviamo ad utilizzare la classe creata:

```
Lista_voti = CListaVoti()
lista_voti.aggiungi_esame()
lista_voti.aggiungi_esame()
lista_voti.aggiungi_esame()
lista_voti.mostra_nome_esame_voto_piu_alto()
```

Da Notare sulla Composizione

Come abbiamo visto, per creare una relazione di composizione non esiste una sintassi particolare (es istruzioni apposite).

Si ha tale relazione quando un oggetto memorizza al suo interno oggetti di una classe differente, che **non ha senso continuino ad esistere nel caso in cui venga eliminato l'oggetto composto**.

La Relazione di Aggregazione

Si ha una relazione di aggregazione quando un oggetto mette insieme altri oggetti che hanno una vita propria: oggetti che devono continuare ad esistere se l’oggetto che li aggrega viene cancellato.

La Relazione di Aggregazione

Supponete di venire assunti dal gestore di una libreria per espandere il programma che utilizza per gestire la libreria. Quel programma, memorizza:

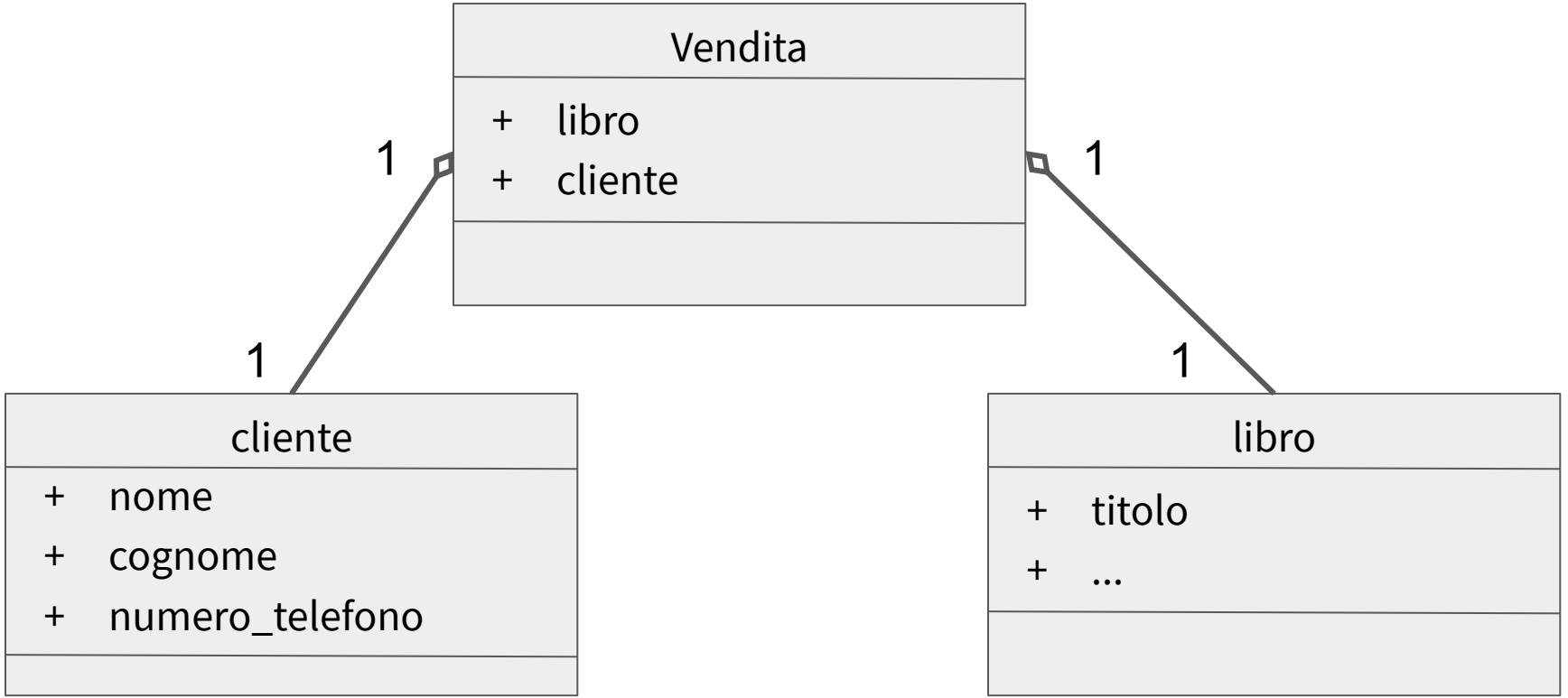
-I dati dei libri dalla libreria nella classe CListaLibri vista precedentemente.

-I dati dei clienti in una classe CListaClienti composta da oggetti di classe CCliente, che hanno gli attributi: nome, cognome, numero_di_telefono.

Il gestore della libreria vi chiede di creare una classe CVendita che permetta di memorizzare, per vendita di un singolo libro, quale libro è stato venduto e a quale cliente è stato venduto.

Questa classe deve poter essere rimossa senza che vengano eliminati i libri venduti o i dati dei clienti e non deve duplicare dati.

La Relazione di Aggregazione



La Relazione di Aggregazione

La classe CVendita memorizzerà un riferimento ad un oggetto di tipo cliente e ad un oggetto libro esistenti.

```
class CVendita:

  def __init__(self, cliente, libro):
     self.cliente = cliente
     self.libro = libro
```

La Relazione di Aggregazione

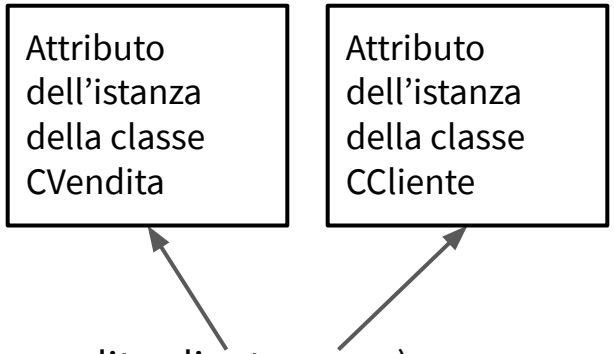
Un esempio di codice che crea un oggetto di classe CVendita è il seguente:

```
titolo = input("inserisci il titolo del libro ")
autore = input("inserisci l'autore del libro ")
editore = input("inserisci l'editore del libro ")
prezzo = float(input("inserisci il prezzo del libro "))
anno_pubblicazione = int(input("inserisci l'anno di pubblicazione del libro "))
oggetto_libro = CLibro(titolo, autore, editore, prezzo, anno_pubblicazione)

oggetto_cliente = CCliente("Anna", "Bianchi", "070889977")

oggetto_vendita = CVendita(oggetto_libro, oggetto_cliente)
```

La Relazione di Aggregazione

Come spiegato, nell'oggetto CVendita vengono memorizzati dei riferimenti ad un oggetto libro e ad un oggetto cliente esistenti. Possiamo quindi accedere agli attributi di quegli oggetti.

oggetto_vendita = CVendita(oggetto_libro, oggetto_cliente)




```
print("Il nome del cliente che ha comprato il libro è ", oggetto_vendita.cliente.nome)
```

Stamperà: Il nome del cliente che ha comprato il libro è Anna

La Relazione di Aggregazione

Nb: Poichè la classe CVendita non memorizza gli oggetti libro e cliente ma solo dei riferimenti ad essi se essi cessassero di esistere questa classe non funzionerebbe più correttamente.

Esercizio sulla Relazione di Aggregazione

Creare una classe chiamata Registrazione Voto, che permetta di memorizzare un riferimento all'oggetto contenente i dati dello studente e un riferimento ad una classe CEsame contenente i voti conseguiti dallo studente in due parziali.

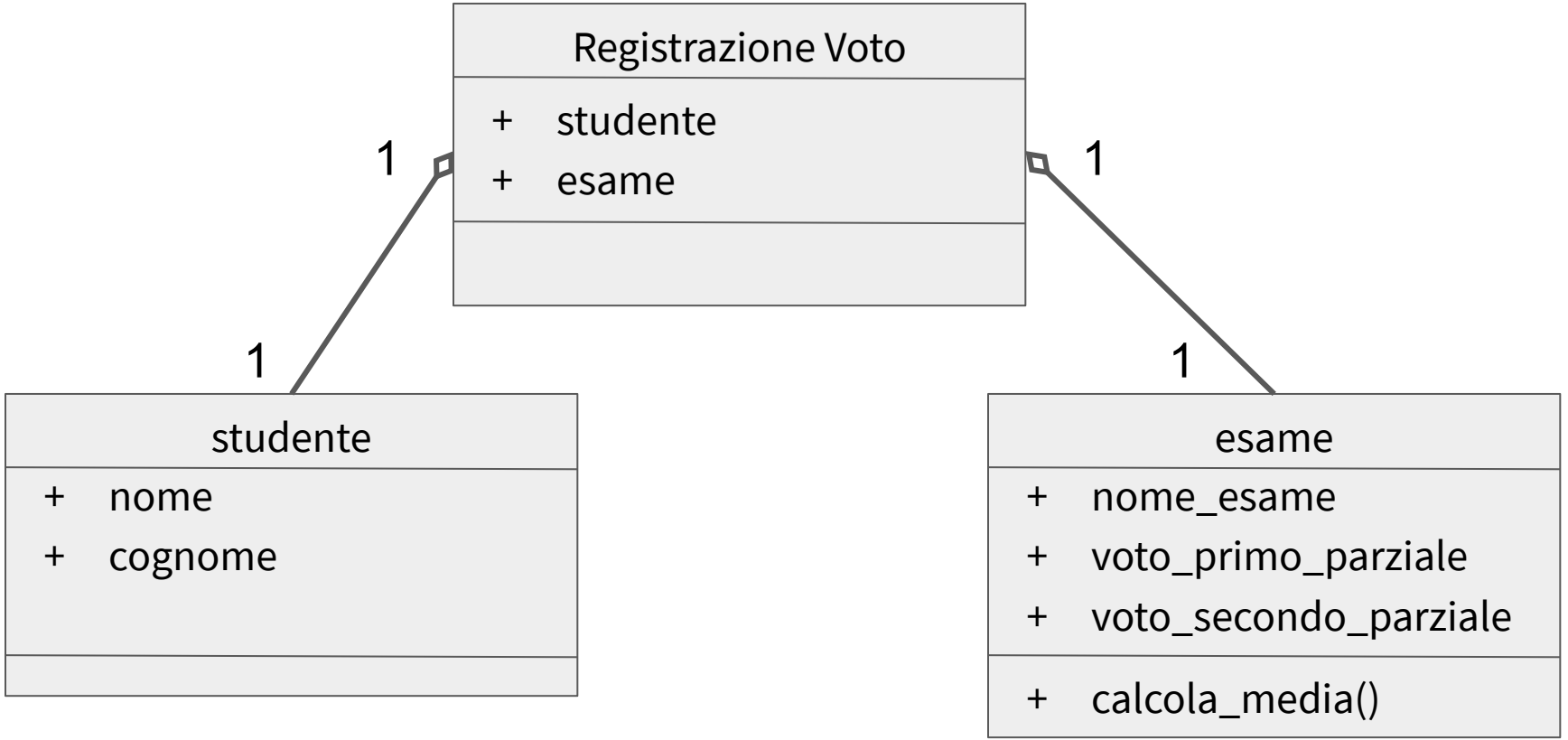
(Il diagramma UML è mostrato nella slide successiva).

Scrivere poi le istruzioni necessarie per:

1) creare un'istanza della classe Registrazione Voto.

2) stampare, facendo riferimento all'istanza creata al punto uno, il nome dell'esame.

Esercizio sulla Relazione di Aggregazione



Esercizio sulla Relazione di Aggregazione

```
class CRegistrazioneVoto:
  def __init__(self, studente, esame):
    self.studente = studente
    self.esame = esame
```

Nella cartella con le soluzioni trovate il file dove i valori degli attributi vengono presi fatti inserire all'utente.

```
oggetto_studente = CStudente("Anna", "Bianchi")
oggetto_esame = CEsame("LPO", 28, 30)
oggetto_reg_voto = CRegistrazioneVoto(oggetto_studente, oggetto_esame)
print("Il nome dell'esame è ", oggetto_reg_voto.esame.nome_esame)
```

La Relazione di Ereditarietà

Come abbiamo visto delle classi possono essere legate da una relazione di ereditarietà. In questo caso **le classi figlie ereditano attributi e metodi definiti nella classe padre**.

La Relazione di Ereditarietà

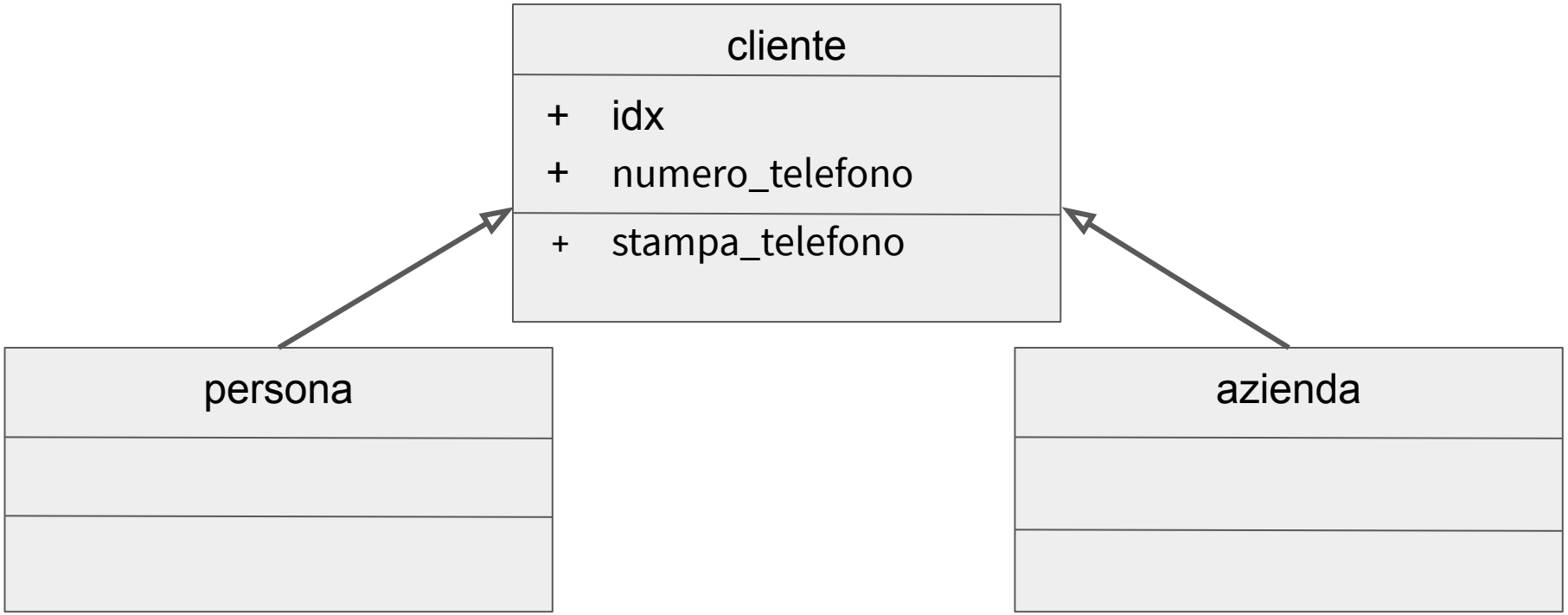
Supponiamo di voler creare tre classi, le classi: persona, azienda e cliente.

Vogliamo che tutte abbiano due attributi: *idx* e *numero_telefono*.

Vogliamo inoltre che entrambe abbiano un metodo chiamato *stampa_telefono* che deve stampare:

Il numero di telefono di <idx> è <numero_telefono>

La Relazione di Ereditarietà



La Relazione di Ereditarietà

Per far si che una classe erediti da un'altra dobbiamo usare la seguente sintassi:

class <nome_classe_figlia> (<nome_classe_padre>):

    ...

Creazione della Classe Padre

Il codice corrispondente al diagramma delle classi visto prima sarà il seguente:

```
class CCliente:
  def __init__(self, idx, numero_telefono):
     self.idx = idx
     self.numero_telefono = numero_telefono

def stampa_telefono(self):
  print("Il numero di telefono di ", self.idx, " è: ", self.numero_telefono)
```

Creazione delle Classi Figlie

In questo esempio le classi figlie **ereditano tutto dalla classe padre**.
Dovremo quindi semplicemente crearle indicando che sono figlie della classe CCliente.

```
class CAzienda(CCliente):
    pass

class CPersona(CCliente):
  pass
```

Poichè le queste classi non devono fare nulla utilizziamo l'istruzione pass. Questa istruzione non fa nulla, serve a riempire un blocco di codice previsto dalla sintassi.

(Se provassimo a rimuovere l'istruzione pass riscontreremmo un errore di sintassi.

Inizializzazione di un Oggetto di Classe CAzienda

Creiamo un oggetto di tipo CAzienda e utilizziamo la funzione *stampa_telefono* che viene ereditata dalla classe padre CCliente.

Esempio:

```
oggetto_azienda = CAzienda("123", "070998899")
oggetto_azienda.stampa_telefono()
```

Questo codice stamperà a schermo:

```
Il numero di telefono di  123  è:  070998899
```

Questo mostra che la classe CAzienda effettivamente ha ereditato il metodo *stampa_telefono* dal padre.

Overriding di un Metodo Definito dalla Classe Padre

Le classi figlie ereditano tutti i metodi dalla classe padre. Tuttavia è **possibile modificare il codice eseguito da un metodo ereditato** dalla classe padre **sovrascrivendo la definizione del metodo** (creando un metodo che abbia lo stesso nome ma un comportamento differente).

Overriding di un Metodo Definito dalla Classe Padre

Supponiamo ora di voler far si che il metodo stampa_telefono stampi:

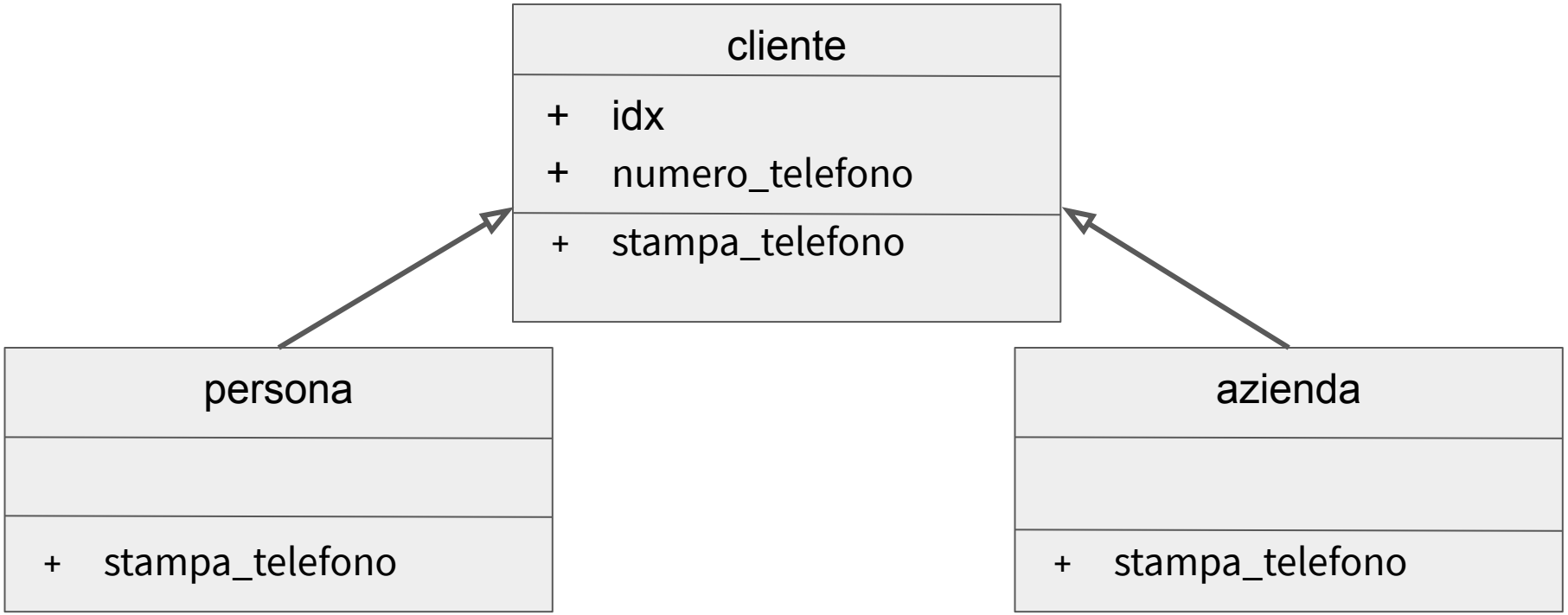
- Per la classe persona:

  Il numero di telefono della persona con <idx> è <numero_telefono>
- Per la classe azienda:

  Il numero di telefono dell'azienda con <idx> è <numero_telefono>

Per far questo supponiamo di voler sovrascrivere nei figli il metodo stampa_telefono definita dalla classe padre.

La Relazione di Ereditarietà



Overriding di un Metodo Definito dalla Classe Padre

Il codice della classe CCliente rimarrà identico a quello visto in precedenza, mentre **cambieranno le definizioni delle classi figlie in quanto devono sovrascrivere il metodo** *stampa_telefono* **in modo da cambiarne il comportamento**.

```
class CAzienda(CCliente):

  def stampa_telefono(self):
    print("Il numero di telefono dell'azienda con ", self.idx, " è: ",
        self.numero_telefono)
```

Overriding di un Metodo Definito dalla Classe Padre

```
class CPersona(CCliente):

  def stampa_telefono(self):
    print("Il numero di telefono della persona con ", self.idx, " è: ",
        self.numero_telefono)
```

Overriding di un Metodo Definito dalla Classe Padre

Proviamo ora a creare una oggetto di classe CAzienda:

```
oggetto_azienda = CAzienda("123", "070998899")
oggetto_azienda.stampa_telefono()
```

Stamperà a schermo:
Il numero di telefono dell'azienda con 123 è: 070998899

Come vediamo quello che viene richiamato è il metodo *stampa_telefono* definito nella classe CAzienda e non più nella classe CCliente.

La Relazione di Ereditarietà

Supponiamo un negoziante che ha come clienti persone e aziende ci abbia commissionato di creare le classi necessarie per memorizzare i seguenti dati di un singolo cliente:

Per le persone: idx, nome, cognome, anno di nascita, numero di telefono.

Per le aziende: idx, partita iva, numero di telefono.

Dove idx è un identificativo univoco assegnato dall'azienda ai suoi clienti.

Le classi create dovranno avere anche un metodo che permette di stampare:

Per le persone:

Il numero di telefono della persona con idx <idx> è <numero_telefono>

Per le aziende:

Il numero di telefono dell' azienda con idx <idx> è <numero_telefono>

La Relazione di Ereditarietà

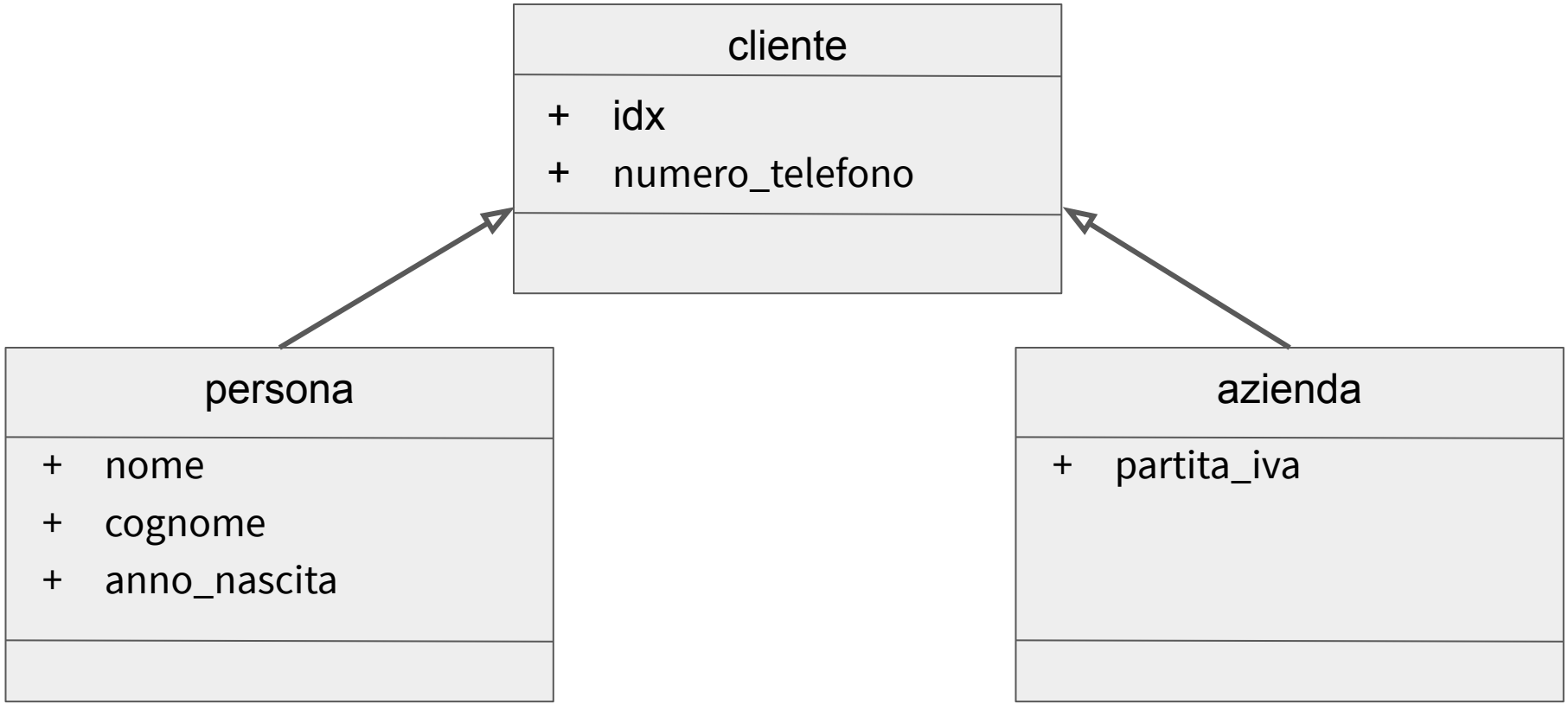
Gli attributi necessari sono differenti a seconda del tipo di cliente quindi dovremo avere due classi (una per le persone e una per le aziende).

Notiamo anche che hanno degli attributi in comune:

Per le persone: idx, nome, cognome, anno di nascita, numero di telefono.
Per le aziende: idx, partita iva, numero di telefono.
Dove idx è un identificativo univoco assegnato dall'azienda ai suoi clienti.

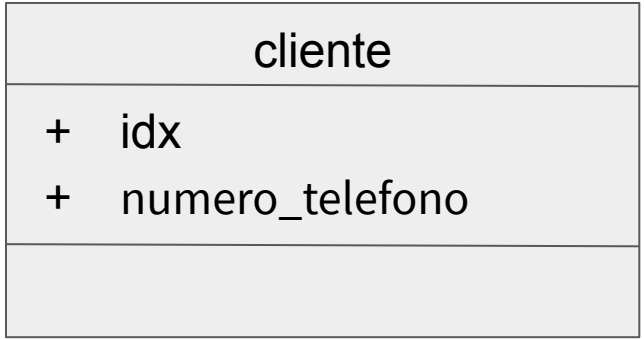
Quindi ha senso creare una terza classe "cliente" che sarà la classe padre.

La Relazione di Ereditarietà



La Classe Padre CCliente

```
class CCliente:
  def __init__(self, idx, numero_telefono):
    self.idx = idx
    self.numero_telefono = numero_telefono
```



La Classe Figlia CAzienda

La classe figlia CAzienda deve occuparsi di memorizzare solo l'attributo partita_iva in quanto gli altri vengono memorizzati dalla classe padre.
Ci si potrebbe quindi aspettare di poter definire la classe CAzienda così:

```
class CAzienda(CCliente):

  def __init__(self, partita_iva):
    self.partita_iva = partita_iva
```

Questo però non è il modo corretto di definirla...

La Classe Figlia CAzienda

```
class CAzienda(CCliente):

  def __init__(self, partita_iva):
     self.partita_iva = partita_iva
```

La classe CCliente acquisisce questi attributi utilizzando il metodo __init__.

Init è un metodo. Questo codice sovrascrive il metodo __init__ definito nella classe padre. Verrebbe quindi memorizzato solo l'attributo *partita_iva* e non verrebbero memorizzati *idx* e *numero_telefono*.

Richiamare un Metodo della Classe Padre

Poichè abbiamo sovrascritto il metodo __init__, se vogliamo sfruttare il fatto che il metodo __init__ della classe padre (CCliente) definisce alcuni attributi dobbiamo richiamarlo esplicitamente.

Per richiamare esplicitamente il metodo __init__ della classe padre si utilizza:
super().__init__(<argomento1>..<argomenton>)

Genericamente, per richiamare esplicitamente un metodo della classe padre si può usare la sintassi:
super().<nome_metodo>(<argomento1>..<argomenton>)

Richiamare una Funzione della Classe Padre

Aggiungiamo la chiamata al metodo __init__ della classe padre.

```
class CAzienda(CCliente):

  def __init__(self, partita_iva):
     self.partita_iva = partita_iva
     super().__init__(idx, numero_telefono)
```

Richiamare un Metodo della Classe Padre

NB: Per poter passare all'init della classe padre i valori degli attributi idx e numero_telefono dobbiamo far si che la classe riceva anche questi attributi al momento della creazione dell'oggetto. Dobbiamo quindi aggiungere i relativi parametri nel metodo __init__ della classe figlia.

```
class CAzienda(CCliente):

  def __init__(self, partita_iva, idx, numero_telefono):
     self.partita_iva = partita_iva
     super().__init__(idx, numero_telefono)
```

Richiamare un Metodo della Classe Padre

Quando creiamo un'istanza della classe CAzienda dovremmo quindi passare come argomento sia i valori degli attributi settati dalla classe CAzienda che quelli settati dalla classe padre CCliente.

```
oggetto_azienda = CAzienda("12345678901", "123", "070998899")
print("La partita iva e' ", oggetto_azienda.partita_iva)
print("Il numero di telefono e' ", oggetto_azienda.numero_telefono)
```

Stamperà:
La partita iva e' 12345678901
Il numero di telefono e' 070998899

La Classe CCliente

La classe CPersona sarà molto simile alla classe CAzienda ma definirà attributi differenti:

```
class CPersona(CCliente):

  def __init__(self, nome, cognome, anno_nascita, idx, numero_telefono):

    self.nome = nome
    self.cognome = cognome
    self.anno_nascita = anno_nascita
    super().__init__(idx, numero_telefono)
```

Richiamare un Metodo della Classe Padre

Leggendo vecchi codici Python potreste trovare la sintassi:
super(<nome_classe>, self).__init__(<argomento1>..<argomenton>)

Questa sintassi è equivalente a:
super().__init__(<argomento1>..<argomenton>)

Esercizio sull’ Ereditarietà

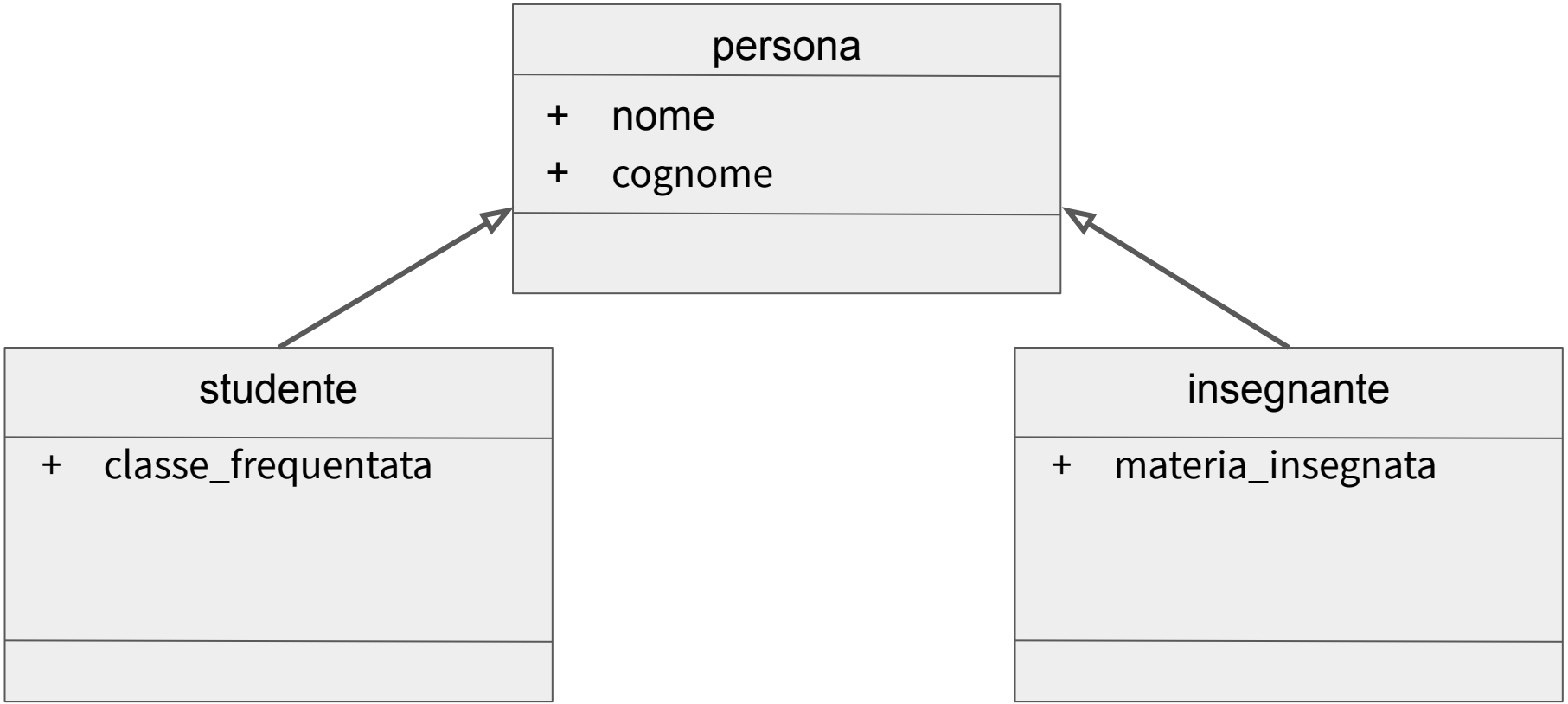
Creare le classi necessarie per memorizzare i dati anagrafici di studenti e insegnanti di una scuola superiore.

Gli attributi da memorizzare sono:

Per gli studenti: matricola, nome, cognome, classe_frequentata.

Per i docenti: nome, cognome, materia_insegnata.

Esercizio sull' Ereditarietà



Esercizio sull' Ereditarietà

La classe padre CCliente:

```
class CCliente:

  def __init__(self, idx, numero_telefono):

      self.idx = idx
      self.numero_telefono = numero_telefono
```

Esercizio sull' Ereditarietà

La classe figlia CAzienda:

```
class CAzienda(CCliente):

  def __init__(self, partita_iva, idx, numero_telefono):
     self.partita_iva = partita_iva
     super().__init__(idx, numero_telefono)
```

Esercizio sull' Ereditarietà

La classe figlia CPersona:

```
class CPersona(CCliente):

  def __init__(self, nome, cognome, anno_nascita, idx, numero_telefono):

      self.nome = nome
      self.cognome = cognome
      self.anno_nascita = anno_nascita
      super().__init__(idx, numero_telefono)
```

Esercizio sull’ Ereditarietà

Creazione di un’istanza della classe CAzienda.

```
oggetto_azienda = CAzienda("12345678901", "123", "070998899")
print("La partita iva e' ", oggetto_azienda.partita_iva)
print("Il numero di telefono e' ", oggetto_azienda.numero_telefono)
```

Stamperà:

```
La partita iva e'  12345678901
Il numero di telefono e'  070998899
```