I sistemi, i modelli, la OOP (object-oriented_programming)

Modulo 1:

1. I sistemi
2. I modelli
3. OOP, la programmazione orientata agli oggetti
4. Le funzioni dunder (double underscore)
5. Attributi, metodi pubblici e privati (incapsulamento)
6. Ereditarietà (polimorfismo)
7. Ereditarietà multipla
8. Override (sovrascrittura) di funzioni (polimorfismo)
9. Lista di oggetti creata dinamicamente

1. I sistemi : definizioni ed esempi

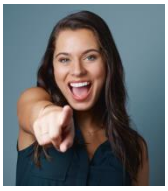
In informatica per sistema si intende, **un insieme di componenti interconnessi che lavorano insieme per raggiungere un determinato obiettivo.** E' una cosa reale che esiste e funziona.
Spesso si presenta come unico oggetto od entità, persona animale o concetto astratto

Esempi in informatica:

- Un personal computer, un cellulare, una stampante, un tablet, un server, una rete di computer, un sistema di gestione database, un sistema di elaborazione di transizione online, ecc..
- Una automobile, un robot, uno scooter, una lavatrice, una nave,un trattore, un distributore di bevande o di carburante , ecc..
- Una persona, un animale
- Un edificio, una scuola, una biblioteca, sistema impiantistico (idraulico, elettrico) ecc..







Riassumendo:

1. Un sistema è composto da diverse parti che hanno funzioni specifiche
2. Le parti sono interconnesse tra loro ed interagiscono per raggiungere l'obiettivo del sistema: ad esempio un'auto è composta da: motore, telaio, organi di trasmissione, carrozzeria, ruote, ecc..
3. Un sistema svolge nel suo complesso una funzione specifica: ad esempio l'auto trasporta le persone, il camion persone e merci, la stampante trasferisce le informazioni su carta

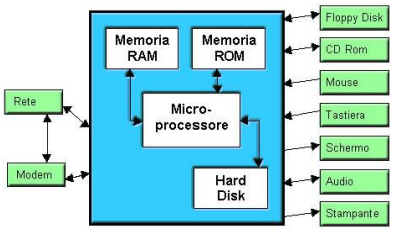
2. I modelli

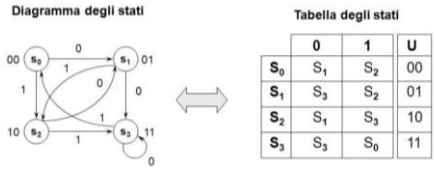
Un modello è una rappresentazione semplificata di un sistema :
Esempi di modelli possono essere:

- Un diagramma, una tabella, un grafico, un grafo, una piantina di un edificio
- Un insieme di equazioni matematiche

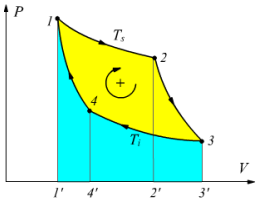
- Un programma (software) con simulazione al computer
- Un disegno al cad, una simulazione al cad
- Un database, un modello OOP (classi ed oggetti)

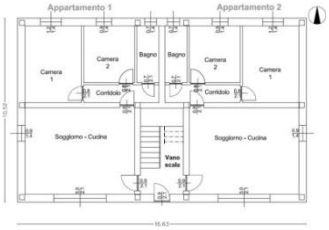
L'obiettivo di un modello è di aiutarci a capire come funziona un sistema, a progettarlo, o a prevederne il comportamento



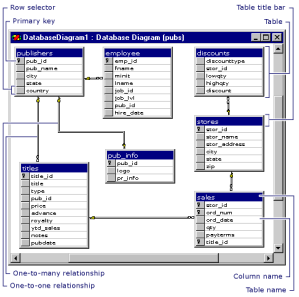


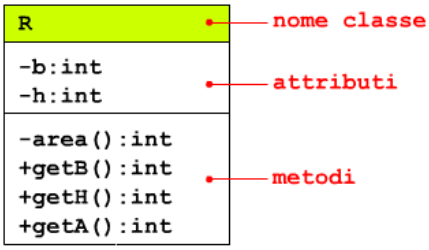
	0	1	U
S_0	S_1	S_2	00
S_1	S_3	S_2	01
S_2	S_1	S_3	10
S_3	S_3	S_0	11





$$\begin{cases} y = 2x + 1 \\ 3x + y = 0 \end{cases}$$





3. OOP (Object oriented programming)

Uno dei punti di forza principali della programmazione orientata agli oggetti (OOP) è proprio la sua capacità di modellare il mondo reale in modo intuitivo e naturale.

L'acronimo OOP sta per **Object-Oriented Programming**, che in italiano si traduce con **Programmazione Orientata agli Oggetti**.

Ecco una spiegazione più dettagliata:

- **Programmazione**: Si riferisce al processo di scrittura di codice per creare un programma per computer.
- **Orientata agli Oggetti**: Indica un paradigma di programmazione in cui il software è progettato organizzando "oggetti" che contengono sia dati (attributi) che codice (metodi).

In pratica, l'OOP è un modo di organizzare il codice in modo che sia più modulare, riutilizzabile e facile da mantenere. Si basa su concetti chiave come:

- **Classi**: Modelli per la creazione di oggetti.
- **Oggetti**: Istanze di classi, che rappresentano entità del mondo reale o concetti astratti.
- **Incapsulamento**: Raggruppamento di dati e metodi all'interno di un oggetto.
- **Ereditarietà**: Possibilità per una classe di ereditare proprietà e metodi da un'altra classe.
- **Polimorfismo:** Possibilità per oggetti di classi diverse di rispondere allo stesso metodo in modi diversi.

Come funziona:

- **Classi**: Le classi fungono da "progetti" o "modelli" per gli oggetti. Definiscono le caratteristiche (attributi) e i comportamenti (metodi) che gli oggetti di quel tipo avranno.




- **Oggetti**: Gli oggetti sono "istanze" delle classi. Sono entità concrete che possiedono le caratteristiche e i comportamenti definiti dalla loro classe.

Esempi pratici:

1. **Modellare una persona**:
   - **Classe**: Persona
   - **Attributi**: nome, cognome, età, indirizzo
   - **Metodi**: cammina(), parla(), mangia()
   - **Oggetti**: persona1 (Mario Rossi), persona2 (Laura Bianchi)
2. **Modellare un'automobile:**
   - **Classe**: Automobile
   - **Attributi**: marca, modello, colore, numero_di_porte
   - **Metodi**: accelera(), frena(), gira()
   - **Oggetti**: auto1 (Fiat Panda), auto2 (Ford Focus)
3. **Modellare un animale:**
   - **Classe**: Animale
   - **Attributi**: specie, razza, età, colore
   - **Metodi**: mangia(), dorme(), emette_suono()
   - **Oggetti**: animale1 (cane), animale2 (gatto)
4. **Modellare una figura geometrica:**
   - **Classe**: Cerchio
   - **Attributi**: raggio, colore
   - **Metodi**: calcola_area(), calcola_circonferenza()
   - **Oggetti**: cerchio1, cerchio2

Vantaggi della modellazione OOP:

- **Rappresentazione realistica**: L'OOP consente di rappresentare entità del mondo reale in modo fedele, con le loro caratteristiche e interazioni.
- **Modularità**: Gli oggetti sono unità indipendenti, il che rende il codice più facile da organizzare e gestire.
- **Riutilizzabilità**: Le classi possono essere riutilizzate per creare più oggetti, riducendo la duplicazione del codice.
- **Manutenibilità**: Le modifiche a una classe si riflettono automaticamente su tutti gli oggetti creati da essa.

In conclusione, l'OOP offre un potente strumento per modellare la realtà in modo efficace e creare software più robusto e flessibile.

Definizione di una classe

In Python, una classe viene definita utilizzando la parola chiave class, seguita dal nome della classe e da due punti. Il corpo della classe contiene attributi (variabili) e metodi (funzioni) che definiscono il comportamento degli oggetti creati da quella classe.

Esempio: classe Rettangolo

Ecco un esempio di come definire una classe Rettangolo in Python:

```
class Rettangolo:
    def __init__(self, base, altezza):
        self.base = base
        self.altezza = altezza

    def calcola_area(self):
        return self.base * self.altezza

    def calcola_perimetro(self):
        return 2 * (self.base + self.altezza)

# Creazione di un oggetto Rettangolo
rettangolo1 = Rettangolo(5, 10)

# Calcolo dell'area e del perimetro
area = rettangolo1.calcola_area()
perimetro = rettangolo1.calcola_perimetro()

# Stampa dei risultati
print(f"Area del rettangolo: {area}")
print(f"Perimetro del rettangolo: {perimetro}")
```

In questo esempio:

- **class Rettangolo:**: Definisce una classe chiamata Rettangolo.
- **__init__(self, base, altezza):**: È il costruttore della classe. Viene chiamato quando viene creato un nuovo oggetto Rettangolo. Il parametro self si riferisce all'oggetto stesso, mentre base e altezza sono i parametri per inizializzare gli attributi dell'oggetto.
- **self.base = base e self.altezza = altezza**: Assegnano i valori dei parametri base e altezza agli attributi dell'oggetto.
- **calcola_area(self): e calcola_perimetro(self):**: Sono metodi che calcolano rispettivamente l'area e il perimetro del rettangolo.
- **rettangolo1 = Rettangolo(5, 10)**: Crea un nuovo oggetto Rettangolo con base 5 e altezza 10.
- **area = rettangolo1.calcola_area() e perimetro = rettangolo1.calcola_perimetro()**: Chiamano i metodi calcola_area() e calcola_perimetro() sull'oggetto rettangolo1 per ottenere i risultati.
- **print(...)**: Stampa i risultati.

Altri aspetti importanti

- **Ereditarietà**: Python supporta l'ereditarietà, che consente di creare nuove classi basate su classi esistenti.
- **Incapsulamento**: Python fornisce meccanismi per l'incapsulamento, che consente di nascondere gli attributi e i metodi interni di una classe.
- **Polimorfismo**: Python supporta il polimorfismo, che consente di utilizzare oggetti di classi diverse in modo intercambiabile.

Spero che questo esempio ti sia utile per capire come definire una classe in Python. Se hai altre domande, non esitare a chiedere.

4. Le funzioni dunder

Le funzioni "dunder" (o metodi magici) in Python sono metodi speciali che iniziano e finiscono con un doppio underscore (__). Sono utilizzati per implementare comportamenti specifici e personalizzare il modo in cui gli oggetti della tua classe interagiscono con il linguaggio Python.

Scopo delle funzioni dunder

- **Sovrascrittura di operatori**: consentono di definire il comportamento degli operatori aritmetici (+, -, *, /), di confronto (==, !=, <, >), e di altri operatori per gli oggetti della tua classe.

- **Personalizzazione del comportamento degli oggetti**: consentono di definire come gli oggetti della tua classe vengono rappresentati come stringhe, come vengono gestiti in contesti booleani, come vengono indicizzati, e così via.
- **Integrazione con il linguaggio Python**: consentono di far sì che gli oggetti della tua classe si comportino come i tipi built-in di Python, come le liste, i dizionari, e i numeri.

Esempi di funzioni dunder

- __init__(self, ...): il costruttore della classe, chiamato quando viene creato un nuovo oggetto.
- __str__(self): restituisce una rappresentazione in stringa "informale" dell'oggetto, utilizzata da print() e str().
- __repr__(self): restituisce una rappresentazione in stringa "formale" dell'oggetto, utilizzata dall'interprete Python e da repr().
- __add__(self, other): definisce il comportamento dell'operatore +.
- __eq__(self, other): definisce il comportamento dell'operatore ==.
- __len__(self): definisce il comportamento della funzione len().
- __getitem__(self, key): definisce il comportamento dell'operatore di indicizzazione [].

Esempio di implementazione

```
class Frazione:
    def __init__(self, numeratore, denominatore):
        self.numeratore = numeratore
        self.denominatore = denominatore

    def __str__(self):
        return f"{self.numeratore}/{self.denominatore}"

    def __add__(self, altra_frazione):
        nuovo_numeratore = (self.numeratore * altra_frazione.denominatore) +
(altra_frazione.numeratore * self.denominatore)
        nuovo_denominatore = self.denominatore * altra_frazione.denominatore
        return Frazione(nuovo_numeratore, nuovo_denominatore)

f1 = Frazione(1, 2)
f2 = Frazione(1, 4)

print(f1)  # Output: 1/2
print(f2)  # Output: 1/4

f3 = f1 + f2
print(f3)  # Output: 6/8
```

In questo esempio, la classe Frazione implementa le funzioni dunder __init__, __str__ e __add__. Questo consente di creare oggetti Frazione, di stamparli in modo leggibile e di sommarli utilizzando l'operatore +.
Le funzioni dunder sono uno strumento potente per personalizzare il comportamento degli oggetti in Python. Ti permettono di scrivere codice più espressivo e di far sì che le tue classi si integrino perfettamente con il linguaggio.

5. Attributi, metodi pubblici e privati (incapsulamento)

In Python la differenza tra attributi e metodi pubblici e privati riguarda la visibilità e l'accessibilità di questi elementi all'interno di una classe e al di fuori di essa.

Attributi e metodi pubblici

- **Accessibilità**: sono accessibili sia all'interno che all'esterno della classe.
- **Convenzione**: in Python, per convenzione, tutti gli attributi e i metodi sono pubblici a meno che non si specifichi diversamente.

Esempio:

```
class Persona:
    def __init__(self, nome, eta):
        self.nome = nome  # Attributo pubblico
        self.eta = eta    # Attributo pubblico

    def saluta(self):      # Metodo pubblico
        print(f"Ciao, sono {self.nome} e ho {self.eta} anni.")

persona1 = Persona("Alice", 30)
print(persona1.nome)  # Accesso all'attributo pubblico
persona1.saluta()      # Chiamata al metodo pubblico
```

Attributi e metodi privati

- **Accessibilità**: sono accessibili solo all'interno della classe.
- **Convenzione**: in Python, si usa la convenzione di anteporre un doppio underscore (__) al nome dell'attributo o del metodo per indicarlo come privato.

Esempio:

```
class Persona:
    def __init__(self, nome, eta, __codice_fiscale):
        self.nome = nome
        self.eta = eta
        self.__codice_fiscale = __codice_fiscale  # Attributo privato

    def __mostra_codice_fiscale(self):  # Metodo privato
        print(f"Il mio codice fiscale è {self.__codice_fiscale}.")

    def saluta(self):
        print(f"Ciao, sono {self.nome} e ho {self.eta} anni.")
        self.__mostra_codice_fiscale()  # Chiamata al metodo privato all'interno
della classe

persona1 = Persona("Alice", 30, "ABC123XYZ")
print(persona1.nome)
persona1.saluta()
# print(persona1.__codice_fiscale)  # Errore: accesso all'attributo privato non
consentito
# persona1.__mostra_codice_fiscale()  # Errore: chiamata al metodo privato non
consentita
```

In questo esempio, __codice_fiscale e __mostra_codice_fiscale sono considerati privati. Non è possibile accedervi direttamente dall'esterno della classe. Tuttavia, è possibile accedervi indirettamente tramite metodi pubblici della classe, come il metodo saluta.

È importante notare che la "privacy" in Python è basata su convenzioni e name mangling, non su restrizioni rigide come in altri linguaggi di programmazione. È comunque possibile accedere agli attributi e metodi "privati" tramite il name mangling, ma è sconsigliato farlo in quanto viola le convenzioni del linguaggio e può rendere il codice più fragile.

Ecco alcuni punti chiave sul name mangling in Python:

- **Scopo**:
  - Prevenire conflitti di nomi in classi derivate (ereditarietà).
  - Fornire una forma di incapsulamento, rendendo gli attributi meno accessibili dall'esterno.
- **Come funziona**:
  - Quando un attributo di classe inizia con __, Python lo rinomina aggiungendo _NomeClasse davanti al nome originale.
  - Ad esempio, un attributo __attributo in una classe MiaClasse viene trasformato in _MiaClasse__attributo.
- **Accesso**:
  - Sebbene il name mangling renda gli attributi più difficili da accedere, non li rende completamente inaccessibili.
  - È ancora possibile accedervi utilizzando il nome modificato.

6. Ereditarietà e polimorfismo

L'ereditarietà è uno dei concetti fondamentali della programmazione orientata agli oggetti (OOP) in Python. Permette di creare nuove classi (chiamate classi derivate, sottoclassi o classi figlie) basate su classi esistenti (chiamate classi base, superclassi o classi padre).
In pratica, una sottoclasse eredita gli attributi e i metodi della sua superclasse, consentendo di riutilizzare il codice e creare una gerarchia di classi con relazioni "è-un".
Ecco i vantaggi principali dell'ereditarietà:

- **Riutilizzo del codice**: Si evita di riscrivere codice già esistente, promuovendo la riusabilità e riducendo la ridondanza.
- **Estensibilità**: Si possono aggiungere nuovi attributi e metodi alle sottoclassi, estendendo le funzionalità delle superclassi.
- **Manutenibilità**: Le modifiche alla superclasse si riflettono automaticamente sulle sottoclassi, semplificando la manutenzione del codice.
- **Polimorfismo**: L'ereditarietà è alla base del polimorfismo, che consente di trattare oggetti di classi diverse in modo uniforme.

Esempio:
Immagina di voler creare un sistema per gestire diversi tipi di veicoli. Potresti iniziare creando una classe base Veicolo con attributi e metodi comuni a tutti i veicoli, come marca, modello e accelera(). Poi, potresti creare sottoclassi come Auto e Moto che ereditano da Veicolo e aggiungono attributi e metodi specifici, come numero_porte per Auto e tipo_motore per Moto.
Ecco un esempio di codice:

```
class Veicolo:
    def __init__(self, marca, modello):
        self.marca = marca
        self.modello = modello

    def accelera(self):
        print("Il veicolo sta accelerando.")

class Auto(Veicolo):
    def __init__(self, marca, modello, numero_porte):
        super().__init__(marca, modello)
        self.numero_porte = numero_porte

    def apri_portiere(self):
        print("Portiere aperte.")

class Moto(Veicolo):
    def __init__(self, marca, modello, tipo_motore):
        super().__init__(marca, modello)
        self.tipo_motore = tipo_motore
```

```
    def impenna(self):
        print("La moto si impenna.")

# Creazione di oggetti
auto = Auto("Fiat", "Panda", 5)
moto = Moto("Honda", "CBR", "In linea")

# Chiamata di metodi
auto.accelera()  # Ereditato da Veicolo
auto.apri_portiere()  # Specifico di Auto

moto.accelera()  # Ereditato da Veicolo
moto.impenna()  # Specifico di Moto

print(auto.marca) # eredita l'attributo marca dalla classe Veicolo
```

In questo esempio, Auto e Moto sono sottoclassi di Veicolo. Ereditano gli attributi marca e modello e il metodo accelera() dalla superclasse. Inoltre, aggiungono i propri attributi e metodi specifici.
Spero che questo esempio ti sia utile. Se hai altre domande, non esitare a chiedere!

7. Ereditarietà multipla

   In python è possibile usare l'ereditarietà multipla. Significa che una classe figlia eredita attributi e metodi da più classi genitore. La sintassi è semplice, si elencano tra parentesi tonde, separate da virgola, le classi genitore, nella definizione della classe figlia.

Spiegazione:

1. **Classe Persona:** Definisce gli attributi comuni a tutte le persone (nome e cognome) e un metodo presentati().
2. **Classe Lavoratore:** Definisce gli attributi e i metodi specifici per un lavoratore (ruolo e stipendio).
3. **Classe Studente:** Definisce gli attributi e i metodi specifici per uno studente (matricola e corso di studio).
4. **Classe LavoratoreStudente:** Eredita da tutte e tre le classi precedenti, combinando gli attributi e i metodi di ciascuna. Nel costruttore __init__(), chiamiamo i costruttori delle superclassi per inizializzare gli attributi ereditati.
5. **Metodo presentati_completo():** Un metodo aggiuntivo che combina le informazioni di tutte e tre le classi.

In questo modo, la classe LavoratoreStudente rappresenta una persona che è sia un lavoratore che uno studente, ereditando le caratteristiche di entrambe le "categorie".

Ecco un esempio di ereditarietà multipla in Python con una superclasse Persona:

```
class Persona:
    def __init__(self, nome, cognome):
        self.nome = nome
        self.cognome = cognome

    def presentati(self):
        return f"Mi chiamo {self.nome} {self.cognome}."

class Lavoratore:
    def __init__(self, ruolo, stipendio):
        self.ruolo = ruolo
        self.stipendio = stipendio
```

```
    def dettagli_lavoro(self):
        return f"Ruolo: {self.ruolo}, Stipendio: {self.stipendio}"

class Studente:
    def __init__(self, matricola, corso_di_studio):
        self.matricola = matricola
        self.corso_di_studio = corso_di_studio

    def dettagli_studio(self):
        return f"Matricola: {self.matricola}, Corso di studio:
{self.corso_di_studio}"

class LavoratoreStudente(Persona, Lavoratore, Studente):
    def __init__(self, nome, cognome, ruolo, stipendio, matricola,
corso_di_studio):
        Persona.__init__(self, nome, cognome)
        Lavoratore.__init__(self, ruolo, stipendio)
        Studente.__init__(self, matricola, corso_di_studio)

    def presentati_completo(self):
        return f"{self.presentati()} {self.dettagli_lavoro()}
{self.dettagli_studio()}"

# Esempio di utilizzo
persona_lavoratore_studente = LavoratoreStudente("Mario", "Rossi",
"Sviluppatore", "2000€", "12345", "Informatica")
print(persona_lavoratore_studente.presentati_completo())
```

8. Override (sovrascrittura) di funzioni (polimorfismo)

Questo è un esempio di come sovrascrivere (override) un metodo in Python utilizzando una gerarchia di classi per figure geometriche.

Concetti chiave

- **Ereditarietà:** Una classe "figlia" eredita attributi e metodi da una classe "genitore".
- **Sovrascrittura (Override):** Una classe figlia può fornire una propria implementazione di un metodo già definito nella classe genitore.
- **super():** Permette di chiamare il metodo della classe genitore dalla classe figlia.

Spiegazione

1. **Classe FiguraGeometrica (Classe Genitore):**
   - Definisce i metodi area(), perimetro() e descrizione() con implementazioni di base.
   - Serve come base per le classi di figure specifiche.
2. **Classe Cerchio (Classe Figlia):**
   - Eredita da FiguraGeometrica.
   - Sovrascrive i metodi area() e perimetro() con calcoli specifici per il cerchio.
   - Usa super().descrizione() per estendere il metodo della super classe.
3. **Classe Rettangolo (Classe Figlia):**
   - Eredita da FiguraGeometrica.
   - Sovrascrive i metodi area() e perimetro() con calcoli specifici per il rettangolo.
   - Usa super().descrizione() per estendere il metodo della super classe.
4. **Esempio di utilizzo:**
   - Crea istanze di Cerchio e Rettangolo.
   - Chiama i metodi sovrascritti per calcolare area e perimetro.

In questo modo, puoi creare una gerarchia di classi per diverse figure geometriche, riutilizzando il codice comune nella classe genitore e personalizzando il comportamento nelle classi figlie.

Esempio di codice

```
import math

class FiguraGeometrica:
    def __init__(self, nome):
        self.nome = nome

    def area(self):
        return "Area non definita per la figura generica."

    def perimetro(self):
        return "Perimetro non definito per la figura generica."

    def descrizione(self):
        return f"Figura geometrica: {self.nome}"

class Cerchio(FiguraGeometrica):
    def __init__(self, raggio):
        super().__init__("Cerchio")
        self.raggio = raggio

    def area(self):
        return math.pi * self.raggio**2

    def perimetro(self):
        return 2 * math.pi * self.raggio

    def descrizione(self):
      return f"{super().descrizione()}, raggio: {self.raggio}"

class Rettangolo(FiguraGeometrica):
    def __init__(self, base, altezza):
        super().__init__("Rettangolo")
        self.base = base
        self.altezza = altezza

    def area(self):
        return self.base * self.altezza

    def perimetro(self):
        return 2 * (self.base + self.altezza)

    def descrizione(self):
      return f"{super().descrizione()}, base: {self.base}, altezza:
{self.altezza}"

# Esempio di utilizzo
cerchio = Cerchio(5)
rettangolo = Rettangolo(4, 6)

print(cerchio.descrizione())
print(f"Area del cerchio: {cerchio.area()}")
print(f"Perimetro del cerchio: {cerchio.perimetro()}")

print(rettangolo.descrizione())
print(f"Area del rettangolo: {rettangolo.area()}")
```

```
print(f"Perimetro del rettangolo: {rettangolo.perimetro()}")
```

9. Liste di oggetti : esempio

```
#Un esempio di come creare dinamicamente una lista di oggetti "Persona"
#in Python utilizzando un ciclo for:
class Persona:
    def __init__(self, nome, cognome, eta):
        self.nome = nome
        self.cognome = cognome
        self.eta = eta

    def __str__(self):
        return f"{self.nome} {self.cognome}, {self.eta} anni"

# Creiamo una lista vuota per contenere gli oggetti Persona
persone = []

# Definiamo i dati delle persone
dati_persone = [
    ("Mario", "Rossi", 30),
    ("Laura", "Verdi", 25),
    ("Giovanni", "Bianchi", 40)
]

# Utilizziamo un ciclo for per creare gli oggetti Persona
#e aggiungerli alla lista
for tupla in dati_persone:
    print(tupla)
    nome = tupla[0]
    cognome = tupla[1]
    eta = tupla[2]
    persona = Persona(nome, cognome, eta)
    persone.append(persona)

# Stampiamo la lista di persone
for persona in persone:
    print(persona)
```

Esempi

https://github.com/albertocesaretti/Python_OOP