Classi

INTRODUZIONE ALLA PROGRAMMAZIONE AD OGGETTI

Programmazione OOP

- Object Orienting Programming
- Attenzione sui DATI da manipolare VS sulle procedure (funzioni) che li manipolano
- Un sistema è visto come un insieme di Oggetti che interagiscono tra loro
- Gli oggetti racchiudono sia i dati che il comportamento

OOP: Classi

- Usate per definire:
  - le caratteristiche di un oggetto: ATTRIBUTI
  - Il suo comportamento: METODI
- Le classi sono ‘’concetti astratte’’: non si riferiscono ad un oggetto specifico, ma servono a definire un modello
- Questo modello può essere usato per creare ISTANZE

OOP: Istanze

- Oggetti creati a partire da una Classe
- Ci tornano un’istanza della classe che si riferisce ad uno specifico oggetto (i cui attributi avranno un valore effettivo)
- Una classe viene usata per creare istanze dello stesso tipo ma con attributi diversi
- E’ possibile usare i metodi definiti nella classe su ogni istanza:
  - istanza.metodo()

OOP: Attributi

- Valori associati all’istanza
- Ad esempio: base e altezza di un rettangolo
- Gli attributi di diverse istanze sono separati
- Per accedere ad un attributo:
  - istanza.attributo

OOP: Metodi

- Descrivono il comportamento dell’oggetto
- Simili alle funzioni ma specifici di ogni classe
- Es.: Sia la classe Rettangolo che la classe cerchio possono definire un metodo calcola_area()
- Possono accedere agli attributi e agli altri metodi dell’istanza
- Per chiamare un metodo:
  - istanza.metodo()

Programmazione OOP

- La OOP apporta diversi VANTAGGI:
  - dati e funzioni sono raggruppati
  - è facile sapere quali operazioni possono essere eseguite sui dati
  - non è necessario importare funzioni per eseguire queste operazioni
  - non è necessario passare i dati alle funzioni
  - le funzioni sono compatibili con i dati
  - è più facile estendere e modificare le funzionalità
  - il codice è più semplice da mantenere
  - è più difficile introdurre bug

ESEMPIO

- PROBLEMA: Dati 100 rettangoli, di cui conosciamo base e altezza, vogliamo sapere area e perimetro di ciascuno
- Creiamo una classe che rappresenta l’oggetto Rettangolo
- Nella classe:
  - Base e altezza: ATTRIBUTI
  - Calcola_area, calcola_perimetro: METODI

ESEMPIO

```
class Rettangolo:
    def __init__(self, base, altezza):
        """Inizializza gli attribute base e altezza."""
        self.base = base
        self.altezza = altezza
    def calcola_area(self):
        """Calcola e torna l'area del rettangolo."""
        return self.base * self.altezza
    def calcola_perimetro(self):
        """Calcola e torna il perimetro del rettangolo."""
        return (self.base + self.altezza) * 2
```

ESEMPIO

#Creiamo un’istanza della classe Rettangolo con base 3 e altezza 5

mio_rettangolo = Rettangolo(3,5)

>>> mio_rettangolo.base # mi tornerà 3

>>> mio_rettangolo.altezza # mi tornerà 5

>>> mio_rettangolo.calcola_area() # mi tornerà 15

>>> mio_rettangolo.calcola_perimetro() # mi tornerà 16

Tocca a voi!

- Create la lista con 100 Rettangoli
- Per ogni Rettangolo stampate base, altezza, area e perimetro
- Suggerimento: Per generare un numero casuale:
  - Importate la libreria random
  - Usate la funzione random.randrange(100)

ESEMPIO

- Creiamo una lista con 100 istanze di Rettangolo e calcoliamone area e perimetro

```
# creo una lista di 100 istanze con valori casuali
rettangoli = [Rettangolo(randrange(100), randrange(100)) for x in range(100)]
# itero la lista e stampo base, altezza, area e perimetro per ogni rettangolo
for rettangolo in rettangoli:
    print('Rettangolo: ', rettangolo.base, rettangolo.altezza)
    print('Area: ', rettangolo.calcola_area())
    print('Perimetro: ', rettangolo.calcola_perimetro())
```

Definire una classe:

- Parola chiave **class,** seguita dal nome della classe, seguita dai due punti (:)
- L’iniziale della classe è, per convenzione, maiuscola
- class Studente:

Definire una classe:

- Aggiungiamo al modello generico i suoi attributi: nome, cognome e corso_di_studi
- Per aggiungere le caratteristiche si usa un metodo speciale, detto inizializzatore o costruttore: __init__(self)
- Quando si creano dei metodi all’interno di una classe il primo parametro è sempre self, overo l’istanza della classe, seguita dagli altri parametri, separati da virgole

Definire una classe:

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

Definire una classe:

- Costruiamo 2 studenti, inizializzando due istanze:

```
studente_uno = Studente(‘’Andrea’’, ‘Sanna’’, ‘’data science’’)
studente_due = Studente(‘’Marta’’, ‘’Melis’’, ‘’informatica’’)
```

Definire una classe:

- Creaiamo un metodo che ci permetta di visualizzare la scheda di ogni studente:

```
def scheda_personale(self):
        return "Scheda Studente\n Nome: {}\n Cognome: {}\n Corso Di Studi: {}.format(self.nome, self.cognome, self. self.corso_di_studi)
```

Variabili di classe

- Sono attributi che vengono **condivisi da tutte le Istanze della Classe**

- Es.: una caratteristica comune a tutti gli Studenti potrebbe essere il numero di Ore di Lezione Settimanali
- Potrebbe essere quindi una buona idea quella di definire una Variabile di Classe chiamata ore_settimanali, in modo che tutte le Istanze abbiano questa proprietà senza bisogno di specificarla ogni volta

```
class Studente:
ore_settimanali = 36 #Variabile Di Classe
```

Variabili di classe

- Per visualizzare il numero di ore settimanali nella scheda_personale, possiamo accedervi in due modi:
  - tramite la Classe:

```
def scheda_personale(self):
return "Scheda Studente\n Nome: {}\n Cognome: {}\n Corso Di Studi: {}\n Ore Settimanali: {}’’.format(self.nome, self.cognome, self. self.corso_di_studi, Studente.ore_settimanali)
```

  - tramite l'Istanza:

```
def scheda_personale(self):
        return "Scheda Studente\n Nome: {}\n Cognome: {}\n Corso Di Studi: {}\n Ore Settimanali: {}’’.format(self.nome, self.cognome, self. self.corso_di_studi, self.ore_settimanali)
```

Variabili di classe: Tocca a voi

- Numero totale di studenti dell’istituto: corpo_studentesco
- Cosa devo modificare?

Eredietarietà

- Immaginiamo ora di voler scrivere sempre un'applicazione per una università, ma che sia in grado di gestire sia studenti che professori
- Tra studenti e insegnanti ci sono chiaramente delle differenze:
  - ad esempio uno studente starà seguendo un indirizzo_di_studio, mentre un insegnante avrà un elenco delle materie che insegna
- Però, sia studenti che insegnanti sono persone con nome, cognome, età, indirizzo, scheda personale
- Quindi ora creeremo: una classe genitore chiamata Persona, in cui specificheremo le caratteristiche comuni ai due, che verranno poi ereditate dalle due sottoclassi figlie, la sottoclasse studente e la sottoclasse insegnante

Eredietarietà

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

	def scheda_personale(self):
		scheda = """Nome: {}, Cognome: {}, Età: {}, Residenza: {}\n’’’’’’.format(self.nome, self.cognome, self.eta,
self.residenza)
		return scheda

	def modifica_scheda(self):
		print("""Modifica Scheda: 1 – Nome 2 – Cognome 3 – Età 4 - Residenza""")
		scelta = input("Cosa Desideri modificare?")
		if scelta == "1":
			self.nome = input("Nuovo Nome--> ")
		elif scelta == "2":
			self. cognome = input("Nuovo Cognome --> ")
		elif scelta == "3":
			self.età = int(input("Nuova età --> "))
		elif scelta == "4":
			self.residenza = input("Nuova Residenza --> ")
```

Eredietarietà

- Per creare le sottoclassi: Studente e Professore basterà: aggiungere delle parentesi, e tra queste parentesi passare il nome della classe genitore da cui voglio ereditare

```
class Studente(Persona):
    pass


class Professore(Persona):
    pass
```

Eredietarietà

- Creo due istanze, una di studente e l’altra di professore e verifico:

```
studente_uno = Studente("Py","Mike",24,"Casa Dello Studente")
professore_uno = Professore("Mario","Rossi",33,"Viale Roma 32")
print(studente_uno.scheda_personale())
print(professore_uno.scheda_personale()
```

Eredietarietà

- Differenzio: aggiungo una variabile profilo per ciascuna delle sottoclassi.
- Al momento delle creazione di ciascun oggetto, vogliamo chiaramente aggiungere:
  - il corso_di_studio seguito per lo Studente
  - un elenco delle materie insegnate per il Professore
- Per fare questo dobbiamo creare una versione personalizzata del metodo __init__ per le due sottoclassi:
  - Utilizziamo la funzione super() per far in modo che nome, cognome, ed età vengano gestiti dal metodo __init__ della classe Persona

Eredietarietà

```
class Studente(Persona):
    profilo = "Studente"
    def __init__(self,nome, cognome, età, residenza, corso_di_studio):
        super().__init__(nome, cognome, età, residenza) # in PY 3
     Persona.__init__(self, nome, cognome, età, residenza) # in PY 2
        self.corso_di_studio = corso_di_studio

class Insegnante(Persona):
    profilo = "Insegnante"
    def __init__(self,nome, cognome, età, residenza, materie=None):
        super().__init__(nome, cognome, età, residenza) # in PY 3
     Persona.__init__(self, nome, cognome, età, residenza)  #in PY 2
        if materie is None:
            self.materie = []
        else:
            self.materie = materie
```

Eredietarietà

- Devo modificare anche la visualizzazione della scheda personale:
- Creo una versione personalizzata del metodo scheda_personale() per ciascuna delle sottoclassi
- Siccome esisteva già un metodo nella superclasse, chiamiamo questo overwriting

Eredietarietà

In Studente:

```
def scheda_personale(self):
scheda = """Profilo:{Studente.profilo} Corso di Studi:{self.corso_di_studio} ***''''''
return super().scheda_personale() + scheda
```

In Professore:

```
def scheda_personale(self):
scheda = f"""Profilo:{Professore.profilo} Materie Professore:{self.materie}***"""
return super().scheda_personale() + scheda
```

Nota: questo è Python 3

Tocca a Voi!

- Creare un metodo per la classe Studente che ci permetta di modificare il corso di studio

- Creare un metodo per la classe Professore che ci consenta di aggiungere delle materie a quelle già insegnate

Soluzione

```
def cambio_corso(self,corso):
    self.corso_di_studio = corso
    print(f"Corso Aggiornato")



def aggiungi_materia(self,nuova):
    if nuova not in self.materie:
        self.materie.append(nuova)
    print("Elenco Aggiornato")
```

Metodi Speciali

- Insieme a __init__ ci sono altri metodi speciali comuni che definisco quando creo una nuova classe:
  - __str__ : metodo che torna una rappresentazione dell'oggetto sotto forma di stringa
  - __repr__ : metodo che torna informazioni utli allo sviluppatore, come il tipo dell'oggeto e il valore degli attributi
  - __iter__: torna un iteratore sull'oggetto
  - __bool__: può essere usato per definire se un oggetto è vero o falso
  - __len__ : può ritornare la lunghezza (numero degli elementi) di un oggetto

- Vengono invocati automaticamente quando eseguiamo: str(istanza) oppure repr(istanza)

Riferimenti

- https://www.python.it/doc/Howtothink/Howtothink-html-it/chap14.htm
- https://www.programmareinpython.it/video-corso-python-programmazione-a-oggetti/01-classi-e-istanze/
- https://www.html.it/pag/15622/classi-in-python/