

Complementi di Programmazione

# Python: Classi e Oggetti

CdL Informatica - Università degli studi di Modena e Reggio Emilia  
AA 2023/2024

Filippo Muzzini

# Reminder OOP

Classe -> definisce variabili e metodi contenuti

Oggetto -> istanza di una classe

Astrazione -> non si guardano i dettagli implementativi (per l'utilizzatore)

Incapsulamento -> protezione degli attributi interni

Ereditarietà -> si ereditano comportamenti (meno codice)

Polimorfismo -> comportamento adattabile

# Classi - definizione

```
class nome_classe:
```

```
    ... metodi, attributi
```

```
class esempio:
```

```
    i = 0
```

```
    def metodo1(self, arg1, agr2):
```

```
        ...
```

Attributo di classe

Metodo

# Classi - definizione

La classe è essa stessa un oggetto (in Python tutto è un oggetto); da non confondere con gli oggetti istanziati dalla classe.

Utilizzando l'oggetto classe è possibile istanziare oggetti di quella classe:

**variabile = NomeClasse()**

Accedere ai contenuti della classe (attributi di classe e metodi):

**NomeClasse.nome\_elemento**

# Classi - attributi di classe

Gli attributi di classe sono attributi che appartengono all'oggetto classe e non agli oggetti istanziati di quella classe.

Sono definiti direttamente nel blocco di codice della classe.

class esempio:

i = 0

a = 'ciao'

# Classi - attributi di classe

Possono essere acceduti direttamente dall'oggetto classe:

**NomeClasse.i**

Ma anche utilizzando gli oggetti istanziati:

**a = NomeClasse()**

**a.i**

# Classi - attributi di classe

Attenzione all'assegnamento.

Se ho un attributo di classe e un oggetto già istanziato

```
NomeClasse.i -> 10
```

```
a = NomeClasse()
```

Se assegno usando l'oggetto

```
a.i = 20 -> ora l'oggetto ha un suo attributo i che nasconde quello di classe
```

# Classi - metodi

class esempio:

```
def metodo1(self, arg1, agr2):
```

```
...
```

Si definiscono con la parola chiave **def** e hanno come primo argomento obbligatorio il riferimento all'oggetto stesso.

Tipicamente lo si chiama **self**

**self** permette di accedere ai metodi/attributi dell'oggetto



# Classi - chiamata dei metodi

```
i = NomeClasse()
```

```
i.metodo1()
```

Quando si chiama il metodo su oggetto **non** si passa il **self**.

Siccome il metodo è chiamato su un oggetto specifico l'interprete è a conoscenza che self sarà riferito a quell'oggetto.

# Classi - costruttore

Il costruttore è un metodo speciale che inizializza l'oggetto istanziato.

Deve avere come nome `__init__`

```
class NomeClasse():
```

```
    def __init__(self, arg1, arg2):
```

```
        ...
```

# Classi - costruttore

Nel costruttore si inizializzano gli attributi dell'oggetto (non quelli di classe).

Lo si fa attraverso **self**.

```
class NomeClasse():  
    def __init__(self, arg1, arg2):  
        self.a = 10
```

# Classi - costruttore

Il costruttore viene chiamato automaticamente quando viene istanziato l'oggetto.

```
a = NomeClasse()
```

# Classi - metodi static e class

Una classe può avere metodi che possono essere chiamati sulla classe stessa invece che sull'istanza.

metodi statici **@staticmethod**

metodi class **@classmethod**

# Classi - metodi static e class

metodi statici **@staticmethod**

```
class NomeClasse:
```

```
    @staticmethod
```

```
    def static_method(x):
```

```
        print(x)
```

metodi class **@classmethod**

```
class NomeClasse:
```

```
    @classmethod
```

```
    def class_method(self, x):
```

```
        print(x)
```

# Classi - metodi static e class

metodi statici **@staticmethod**

```
class NomeClasse:
```

```
    @staticmethod
```

```
    def static_method(x):
```

```
        print(x)
```

metodi class **@classmethod**

```
class NomeClasse:
```

```
    @classmethod
```

```
    def class_method(self, x):
```

```
        print(x)
```

ATTENZIONE! self in questo caso è  
l'oggetto classe e non l'oggetto istanziato

# Classi - metodi static e class

I metodi statici e class vengono chiamati come altri metodi ma utilizzando l'oggetto class.

**Questo implica che non vi è un oggetto istanziato e quindi tali metodi potranno accedere solo agli attributi di classe.**



# Classi - metodi static e class

```
class Date(object):
```

```
    @classmethod
```

```
    def from_string(cls, date_as_string):
```

```
        day, month, year = map(int, date_as_string.split('-'))
```

```
        date1 = cls(day, month, year)
```

```
        return date1
```

← viene istanziata la classe

```
    @staticmethod
```

```
    def is_date_valid(date_as_string):
```

```
        day, month, year = map(int, date_as_string.split('-'))
```

```
        return day <= 31 and month <= 12 and year <= 3999
```

# Classi - ciclo di inizializzazione

Quando si istanzia un'oggetto

```
a = NomeClasse()
```

l'interprete chiama due metodi:

- il primo per creare l'oggetto
- il secondo per inizializzarlo

# Classi - ciclo di inizializzazione

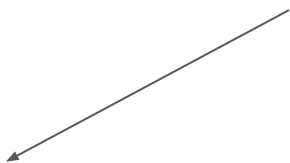
Quando si istanzia un'oggetto

```
a = NomeClasse()
```

l'interprete chiama due metodi:

- il primo per creare l'oggetto
- **il secondo per inizializzarlo**

è il costruttore `__init__` che abbiamo visto



# Classi - ciclo di inizializzazione

Quando si istanzia un'oggetto

```
a = NomeClasse()
```

l'interprete chiama due metodi:

- il primo per creare l'oggetto
- **il secondo per inizializzarlo**

è il costruttore `__init__` che abbiamo visto

Non restituisce nulla in quanto l'oggetto esiste già (è self) e viene solamente inizializzato

# Classi - ciclo di inizializzazione

- **il primo per creare l'oggetto**

E' il metodo `__new__` che tipicamente non viene definito in quanto si usa quello di default.

Tale metodo prende in ingresso una classe e si occupa di restituire un oggetto di quella classe (non ancora inizializzato).

# Classi - ciclo di inizializzazione

```
class class_name:
```

```
    def __new__(cls, *args, **kwargs):
```

```
        print( "ciao io sono __new__" )
```

```
        return object.__new__(cls)
```

# Classi - ciclo di inizializzazione

```
class class_name:
```

```
    def __new__(cls, *args, **kwargs):
```

```
        print( "ciao io sono __new__" )
```

```
        return object.__new__(cls)
```

restituisce l'oggetto che verrà poi passato ad `__init__`

In questo esempio si chiama `__new__` della classe madre `object`

# Classi - Metodi/attributi privati

In Python i metodi e gli attributi di una classe sono pubblici.

Per rendere privata un'entità si agisce sul nome:

**Le entità che iniziano con 2 underscore `__` e terminano con massimo un underscore `_` vengono considerati privati**

```
class Prova:
```

```
    def __init__(self):
```

```
        self.__priv = 'privato'
```



# Classi - Metodi/attributi privati

In realtà è possibile accedere anche alle entità private.

Per nasconderle Python usa un trucco:

rinomina le entità private come **`_NomeClasse__NomeEntità`**

In questo modo non è possibile accederle utilizzando **`.NomeEntità`**

ma è comunque possibile accederle dall'esterno

# Classi - Metodi speciali

Inoltre vi sono metodi speciali che **iniziano e terminano con 2 underscore** \_\_

Essi possono essere definiti e hanno una semantica speciale.

Vengono chiamati in automatico dall'interprete in certe situazioni.

Es. `__init__` viene chiamato in automatico per inizializzare un oggetto.

# Classi - Metodi speciali

- **`__init__`** costruttore
- **`__repr__`** rappresentazione dell'oggetto
- **`__getattr__`** per emulare l'accesso ad una attributo: `a.x --> a.__getattr__('x')`
- **`__getitem__`** per emulare un tipo contenitore (tipo le liste): `a[x] -> a.__getitem__('x')`
- **`__add__`**, **`__mul__`**, etc. per emulare somme, moltiplicazioni etc.
- **`__call__`** l'oggetto può essere chiamato come una funzione
- **`__str__`** converte un oggetto in una stringa
- **`__del__`** distrugge la classe ( metodo distruttore )
- **`__{ eq , gt, ge, lt, le }__`** verifica se un valore è { uguale, ..., ... } ad un altro
- **`__setitem__`** operatore `[]` in uscita (assegnazione)

# Classi - Metodi speciali

Notare che questi metodi sono associati ad operatori Python ([], +, \*, ecc...)

Questo significa che quando si utilizzano questi operatori in realtà vengono chiamati questi metodi.

E' possibile avere comportamenti specializzati per i vari operatori in base a quale oggetto è coinvolto.

Polimorfismo attraverso l'overloading (sovrascriviamo quello di default)

# Classi - Metodi speciali

Sovrascrivere i metodi speciale può essere molto utile per personalizzare il comportamento di un oggetto.

# Classi - Metodi speciali

Sovrascrivere i metodi speciale può essere molto utile per personalizzare il comportamento di un oggetto.

`__getattr__` e `__setattr__` vengono utilizzati per accedere e settare gli attributi di un oggetto.

Sono i veri metodi utilizzati quando si usa l'operatore `.` e `=`

# Classi - Metodi speciali

Sovrascrivere tali metodi può avere molteplici finalità

- Aggiornare un altro campo in conseguenza di un settaggio

class Prova:

```
def __setattr__(self, campo, valore):
```

```
    if campo == 'saldo':
```

```
        super().__setattr__(campo, valore)
```

```
        self.movimenti.append(valore)
```

NON USARE

self.\_\_setattr\_\_(campo, valore)



# Classi - Metodi speciali

Sovrascrivere tali metodi può avere molteplici finalità

- Aggiornare un altro campo in conseguenza di un settaggio

class Prova:

```
def __setattr__(self, campo, valore):
```

```
    if campo == 'saldo':
```

```
        super().__setattr__(campo, valore)
```

```
        self.movimenti.append(valore)
```

NON USARE  
self.\_\_setattr\_\_(campo, valore)  
  
RICORSIONE!



# Classi - Metodi speciali

Sovrascrivere tali metodi può avere molteplici finalità

- Proteggere l'accesso

```
class Prova:
```

```
    def __getattr__(self, campo):  
        if campo == 'saldo':  
            raise AttributeError('campo non accessibile')  
        else:  
            return super().__getattr__(campo)
```

# Classi - Metodi speciali

```
class Prova:
```

```
    def __getattr__(self, campo):
```

```
        if campo == 'saldo':
```

```
            raise AttributeError('campo non accessibile')
```

```
        else:
```

```
            return super().__getattr__(campo)
```

```
a = Prova()
```

```
a.saldo -> Error!
```

# Classi - Getters and Setters

Tipicamente nei linguaggi ad oggetti si proteggono i dati dell'oggetto.

Si accede ad essi tramite metodi

Tali metodi vengono chiamati

- getters -> per accedere
- setters -> per settare

# Classi - Getters and Setters

Abbiamo già visto che in Python è possibile proteggere (o quasi) un attributo dall'accesso esterno.

Si possono definire metodi per l'accesso al metodo

# Classi - Getters and Setters

```
class Prova:
```

```
    def __init__(self, x):
```

```
        self.__x__ = x
```

```
    def getx(self):
```

```
        return self.__x__
```

```
    def setx(self, x):
```

```
        self.x = __x__
```

```
a = Prova(10)
```

```
print(a.getx())
```

```
a.setx(20)
```

# Classi - Getters and Setters (usando `__getattr__`)

class Prova:

```
    def __init__(self, x):  
        self.__x__ = x  
  
    def __getattr__(self, attr):  
        if attr == 'x':  
            return self.__x__  
  
        else:  
            return super().__getattr__(attr)  
  
    def __setattr__(self, attr, value):  
        if attr == 'x':  
            self.__x__ = value  
  
        else:  
            super().__setattr__(attr, value)
```

```
a = Prova(10)
```

```
print(a.x)
```

```
a.x = 20
```

# Classi - Getters and Setters (usando `__getattr__`)

class Prova:

```
    def __init__(self, x):  
        self.__x__ = x  
  
    def __getattr__(self, attr):  
        if attr == 'x':  
            return self.__x__  
  
        else:  
            return super().__getattr__(attr)  
  
    def __setattr__(self, attr, value):  
        if attr == 'x':  
            self.__x__ = value  
  
        else:  
            super().__setattr__(attr, value)
```

```
a = Prova(10)
```

```
print(a.x)
```

```
a.x = 20
```

Se ho molti attributi l'if diventa ingestibile

# Classi - Getters and Setters @property

```
class Prova:
```

```
    def __init__(self, x):
```

```
        self.__x__ = x
```

```
    @property
```

```
    def x(self):
```

```
        return self.__x__
```

```
    @x.setter
```

```
    def x(self, new_x):
```

```
        self.__x__ = new_x
```

```
a = Prova(10)
```

```
print(a.x)
```

```
a.x = 20
```



# Classi - Getters and Setters @property

il decoratore `@property` è un decoratore built-in che permette di annotare quelli che sono i getters e i setter per un attributo (potrebbe anche non essere in memoria)

con **`@property`** si annota il metodo che deve restituire il valore. Lo si accederà attraverso il nome del metodo (che però verrà trattato come attributo)

con **`@<nome>.setter`** si annota il setter dell'attributo

# Classi - @property non in memoria

```
class Prova:
```

```
    @property
```

```
    def x(self):
```

```
        //accesso ad un DB esterno
```

```
    @x.setter
```

```
    def x(self, new_x):
```

```
        //salvataggio su un DB esterno
```

# Classi - Ereditarietà

Una delle potenzialità dei linguaggi ad oggetti è l'Ereditarietà.

Essa permette di far ereditare i metodi e gli attributi di una classe madre alle classi figlie.

Python permette ciò.

Python permette l'ereditarietà multipla.

# Classi - Ereditarietà

```
class Automobile:
```

```
    def __init__(self):
```

```
        self.posizione = 0
```

```
    def muoviti(self, l):
```

```
        self.posizione += l
```

```
class Fuoristrada(Automobile):
```

```
    def ridotte(self):
```

```
        self.rapporto = 0.01
```

Fuoristrada eredita tutti i  
metodi/attributi di Automobile

# Classi - Ereditarietà

```
class Automobile:
```

```
    def __init__(self):
```

```
        self.posizione = 0
```

```
    def muoviti(self, l):
```

```
        self.posizione += l
```

```
class Fuoristrada(Automobile):
```

```
    def ridotte(self):
```

```
        self.rapporto = 0.01
```

La classe madre viene esplicitata tra parentesi

# Classi - Ereditarietà

```
class Automobile:
```

```
    def __init__(self):
```

```
        self.posizione = 0
```

```
    def muoviti(self, l):
```

```
        self.posizione += l
```

```
class Fuoristrada(Automobile):
```

```
    def ridotte(self):
```

```
        self.rapporto = 0.01
```

```
    def muoviti(self, l):
```

```
        self.posizione += (l*self.rapporto)
```

I metodi possono essere sovrascritti

# Classi - Ereditarietà

```
class Automobile:
```

```
    def __init__(self):
```

```
        self.posizione = 0
```

```
    def muoviti(self, l):
```

```
        self.posizione += l
```

```
class Fuoristrada(Automobile):
```

```
    def ridotte(self):
```

```
        self.rapporto = 0.01
```

```
    def muoviti(self, l):
```

```
        l = l*self.rapporto
```

```
        super().muoviti(l)
```

Per chiamare un metodo della classe madre si usa `super()`

# Classi - Ereditarietà Multipla

```
class Automobile:
```

```
    def __init__(self):
```

```
        self.posizione = 0
```

```
    def muoviti(self, l):
```

```
        self.posizione += l
```

Anfibio eredita da entrambe le classi

```
class Barca:
```

```
    def affonda(self):
```

```
        self.affondato = True
```

```
class Anfibio(Barca, Automobile):
```

```
    pass
```



# Classi - Ereditarietà Multipla

```
class Automobile:
```

```
    def __init__(self):
```

```
        self.posizione = 0
```

```
    def muoviti(self, l):
```

```
        self.posizione += l
```

Anfibio eredita da entrambe le classi

```
class Barca:
```

```
    def affonda(self):
```

```
        self.affondato = True
```

```
class Anfibio(Barca, Automobile):
```

```
    pass
```

# Classi - Ereditarietà Multipla

```
class Automobile:
```

```
    def __init__(self):
```

```
        self.posizione = 0
```

```
    def muoviti(self, l):
```

```
        self.posizione += l
```

Quale muoviti verrà utilizzato?

```
class Barca:
```

```
    def affonda(self):
```

```
        self.affondato = True
```

```
    def muoviti(self, l):
```

```
        if self.affondato:
```

```
            return False
```

```
class Anfibio(Barca, Automobile):
```

```
    pass
```

# Classi - Ereditarietà Multipla

```
class Automobile:
```

```
    def __init__(self):
```

```
        self.posizione = 0
```

```
    def muoviti(self, l):
```

```
        self.posizione += l
```

Python parte dalla classe figlie, se il metodo non è definito lì passa alle classi madri da sinistra a destra.

```
class Barca:
```

```
    def affonda(self):
```

```
        self.affondato = True
```

```
    def muoviti(self, l):
```

```
        if self.affondato:
```

```
            return False
```

```
class Anfibio(Barca, Automobile):
```

```
    pass
```

# Classi - Ereditarietà Multipla

si può verificare l'ordine di risoluzione (Method Resolution Order) accedendo all'attributo `__mro__` o il metodo `mro()`

```
a = Anfibio()
```

```
a.__mro__
```

```
a.mro()
```

# Classi - Ereditarietà Multipla

```
a = Anfibio()
```

```
a.__mro__
```

```
a.mro()
```

`__mro__` è in sola lettura quindi non può cambiare una volta inizializzato.

# Classi - Ereditarietà Multipla

`__bases__` è un attributo che indica le classi base.

è praticamente quello che scriviamo tra parentesi nella definizione della classe

Può essere modificato -> dopo la modifica `__mro__` viene ricalcolato

```
a = Anfibio()
```

```
print(Anfibio.__mro__)
```

```
Anfibio.__bases__ = (Automobile, Barca)
```

```
print(Anfibio.__mro__)
```

# Classi - Ereditarietà Multipla - Costruttore

Se non specificato nella classe figlia, di default viene invocato il costruttore della prima classe madre (e non quello della altre).

Per invocarli entrambi bisogna esplicitarlo

```
class Anfibio(Barca, Automobile):
```

```
    def __init__(self):
```

```
        Barca.__init__(self)
```

```
        Automobile.__init__(self)
```

# Classi - Ereditarietà Multipla - Check classe

- `isinstance(ist, classe)` serve per verificare il tipo di un'istanza di una classe es.  
`isinstance(x,int)`
- `issubclass(x,y)` serve per verificare se x è una sottoclasse di y