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:

- Ereditarietà Multipla
- The Diamond Problem
- Le classi Mixin

Ereditarietà Singola e Multipla

Come sappiamo, la programmazione orientata agli oggetti prevede la relazione di ereditarietà.

L'**ereditarietà** viene definita:

- **singola**, quando una classe eredita da un'altra classe
- **multipla**, quando una classe eredita da più classi.

Solo alcuni linguaggi di programmazione orientati agli oggetti, tra cui Python, permettono di utilizzare la relazione di ereditarietà multipla.

Ereditarietà Multipla in Python

La sintassi per far si che una classe erediti da più classi è:

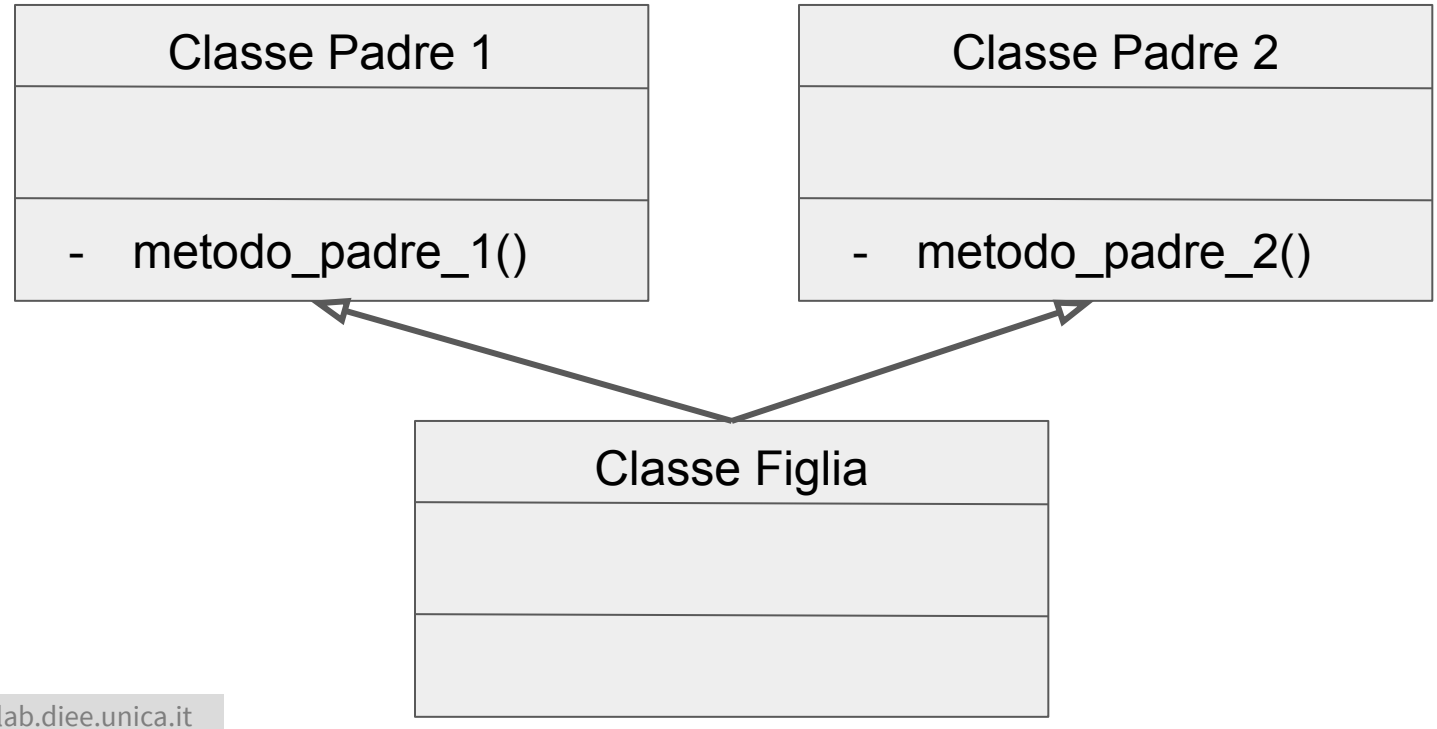
class <nome_classe> (<classe_padre1>, .. <classe_padren>)

    ...

La classe figlia eredita attributi e metodi definiti nelle classi padri.

Ereditarietà Multipla in Python

Supponiamo di voler implementare in Python il seguente prototipo:



Ereditarietà Multipla in Python

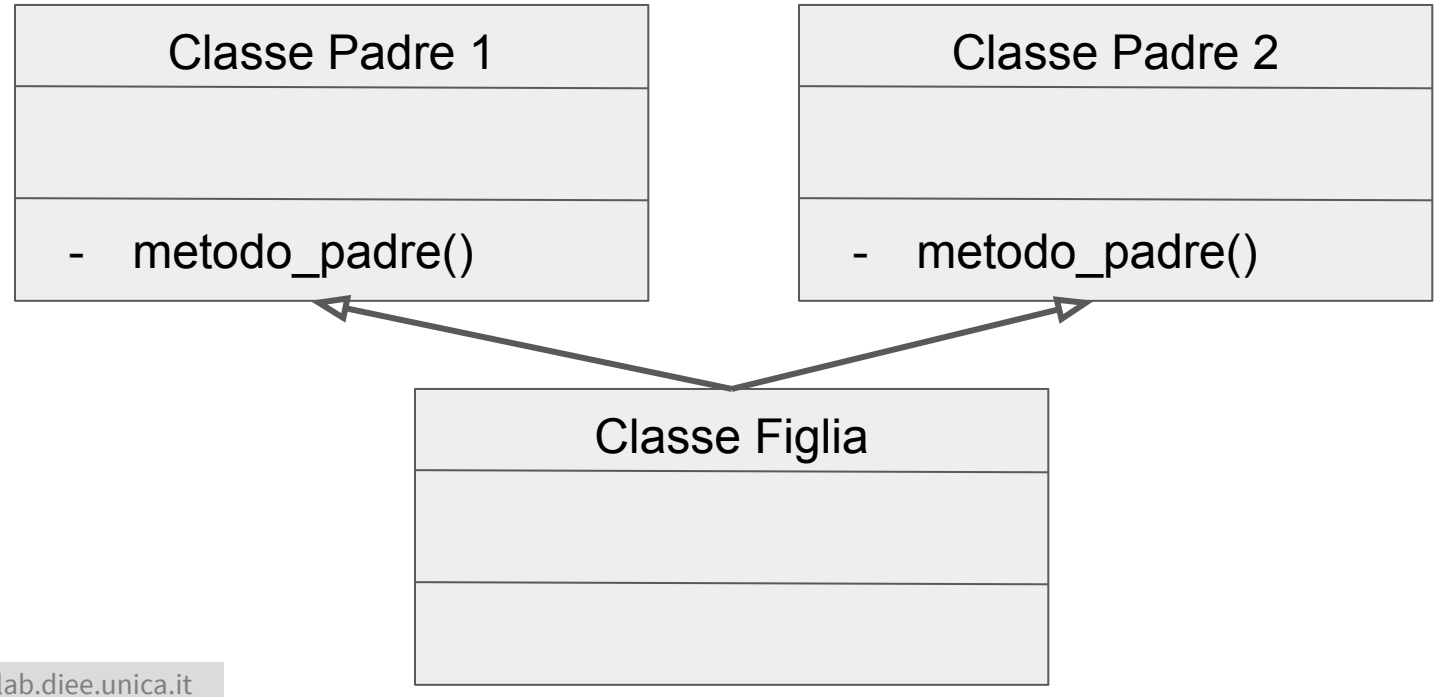
Dovremmo definire la classe figlia come:

```
class CClasseFiglia ( CClassePadre1, CClassePadre2 ):
  pass
```

Le classi padre hanno due metodi con nomi differenti, quindi, la classe figlia erediterà sia il metodo *metodo_padre_1* implementato nella classe padre *CClassePadre1* che *metodo_padre_2* implementato nella classe padre *CClassePadre2*.

Ereditarietà Multipla in Python

Le classi padre potrebbero potenzialmente definire un metodo o un attributo di classe aventi lo stesso nome.



Ereditarietà Multipla in Python

In quel caso la classe figlia eredita il metodo/attributo della classe padre specificata per prima (più a sinistra).

```
class CClasseFiglia ( CClassePadre1, CClassePadre2 ):
  pass
```

Quindi nell'esempio erediterebbe il metodo *metodo_padre* dalla classe *CClassePadre1*.

Ereditarietà Multipla in Python

```
class CClassePadre1:
  def metodo_padre(self):
    print("funzione della classe padre 1")

class CClassePadre2:
  def metodo_padre(self):
    print("funzione della classe padre 2")

class CClasseFiglia(CClassePadre1,CClassePadre2):
  pass
```

Ereditarietà Multipla in Python

oggetto_classe_figlia = CClasseFiglia().metodo_padre()

Stamperebbe:

funzione della classe padre 1

Ereditarietà Multipla in Python

Se invece la classe padre 1 non implementasse “metodo_padre”..

```
class CClassePadre1:
        pass

class CClassePadre2:
  def metodo_padre(self):
     print("funzione della classe padre 2")

class CClasseFiglia(CClassePadre1,CClassePadre2):
  pass
```

Ereditarietà Multipla in Python

oggetto_classe_figlia = CClasseFiglia().metodo_padre()

Stamperebbe:

funzione della classe padre 2

Poichè la prima classe padre specificata non ha il metodo chiamato metodo_padre, la classe figlia erediterebbe il metodo dalla seconda classe padre specificata.

The Diamond Problem

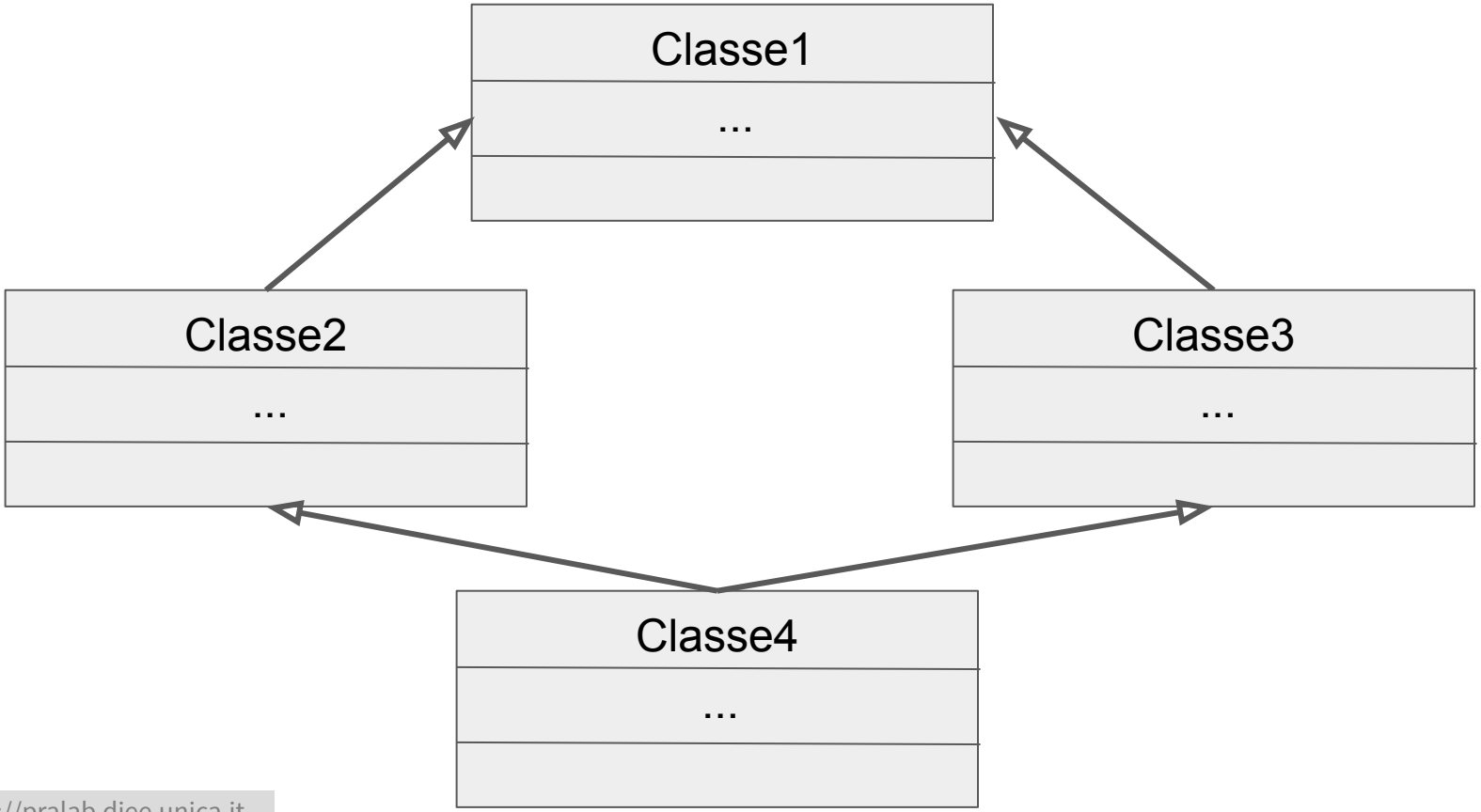
L’utilizzo dell’ereditarietà multipla **può generare problemi** tra i quali quello chiamato: “**the diamond problem”**.

Per capire perchè si verifica supponiamo di avere quattro classi imparentate come mostrato nel diagramma delle classi mostrato nella slide successiva.

Questo tipo di ereditarietà è chiamata “ereditarietà a diamante” per la forma del diagramma delle classi.

Supponiamo che tutte queste classi abbiano una funzione di inizializzazione che deve essere richiamata.

The Diamond Problem



The Diamond Problem

Come implementiamo i metodi __init__ in modo che tutte le classi vengano inizializzate?

Cosa succederebbe se il metodo __init__ di ogni classe si occupasse di definire l'attributo peculiare di quella classe e chiamasse esplicitamente il metodo __init__ della classe padre?

The Diamond Problem

```
class CClasse1:
  def __init__(self):
     print("funzione della classe 1")

class CClasse2(CClasse1):
  def __init__(self):
     print("funzione della classe 2")
     CClasse1.__init__(self)

class CClasse3(CClasse1):
  def __init__(self):
     print("funzione della classe 3")
     CClasse1.__init__(self)
```

The Diamond Problem

```
class CClasse3(CClasse1):
  def __init__(self):
     print("funzione della classe 3")
     CClasse1.__init__(self)

class CClasse4(CClasse2, CClasse3):
  def __init__(self):
     print("funzione della classe 4")
     CClasse2.__init__(self)
     CClasse3.__init__(self)
```

The Diamond Problem

oggetto_classe_4 = CClasse4()

Stamperebbe:
funzione della classe 4
funzione della classe 2
funzione della classe 1
funzione della classe 3
funzione della classe 1

La funzione di inizializzazione della classe 1 è stata richiamata due volte!

Questo potrebbe essere un problema nel caso in cui contenga operazioni computazionalmente costose..

The Diamond Problem

Questo problema potrebbe essere risolto utilizzando la funzione super().

Nel caso dell'ereditarietà multipla la funzione super() non restituisce la classe padre ma la "prossima classe" secondo un ordine definito da Python in modo che metodo della gerarchia venga eseguito una sola volta.

The Diamond Problem

```
class CClasse1:
  def __init__(self):
    print("init della classe 1")

class CClasse2(CClasse1):
  def __init__(self):
    print("init della classe 2")
    super().__init__()

class CClasse3(CClasse1):
  def __init__(self):
    print("init della classe 3")
    super().__init__()
```

The Diamond Problem

```
class CClasse4(CClasse2, CClasse3):
  def __init__(self):
    print("init della classe 4")
    super().__init__()

oggetto_classe_4 = CClasse4()
```

Stamperebbe:
init della classe 4
init della classe 2
init della classe 3
init della classe 1

The Diamond Problem

In questo semplice esempio la funzione super sembra risolvere facilmente il problema.

Tuttavia l'ordine secondo cui i metodi verranno chiamati non è intuitivo.

Inoltre, le cose si complicherebbero notevolmente se i metodi __init__ avessero diversi parametri.

The Diamond Problem

Nel codice di esempio, quando istanziamo un oggetto appartenente a CClasse4, il metodo __init__ della CClasse2 richiama quello della classe CClasse3.

Se invece istanziamo un oggetto di CClasse2, il metodo __init__ di CClasse2 richiamerebbe il metodo __init__ di CClasse1.

The Diamond Problem

oggetto_classe_2 = CClasse2()

Stamperebbe:
init della classe 2
init della classe 1

The Diamond Problem

Questo significa che nell'istruzione:
super().__init__() dovremmo indicare differenti argomenti a seconda del tipo di oggetto che stiamo istanziando.

Questo si potrebbe gestire ma renderebbe il codice estremamente complicato e poco comprensibile!

Ereditarietà Multipla

L'ereditarietà multipla è **generalmente sconsigliata a meno che non si abbia un'ottima ragione per utilizzarla**.

Dal testo di riferimento di questo corso:

"As a humorous rule of thumb, if you think you need multiple inheritance, you're probably wrong, but if you know you need it, you might be right."

Classi Mixin

Una buona ragione per utilizzare l'ereditarietà multipla è quando si hanno classi differenti con alcune funzionalità in comune.

(Definire i metodi in comune in entrambe le classi porterebbe alla duplicazione del codice di definizione dei metodi).

Classi Mixin

In questo caso, si può creare una classe che implementa le funzionalità in comune tra le diverse classi.

Questa classe non è fatta per essere istanziata ma per essere ereditata da tutte le classi che devono fornire quelle funzionalità.

Una classe avente le caratteristiche elencate sopra viene chiamata **mixin**.

Classi Mixin

Ad esempio, supponete di avere creato nel vostro programma due classi chiamate CStringa e CLista che memorizzano al loro interno rispettivamente una stringa e una lista e implementano alcuni metodi a voi utili.

Supponete di voler aggiungere ad entrambe il metodo chiamato *conta_elementi* che conta il numero di caratteri presenti nella stringa o il numero di elementi presenti nella lista.

Possiamo creare una classe mixin chiamata CContaElementiMixin che implementa il metodo *conta_elementi*.

Ciclo For sulle Stringhe

Per utilizzare una sintassi compatta possiamo sfruttare il fatto che **il ciclo for funziona** sia sulle liste, sia **sulle stringhe.** Nel caso delle stringhe la variabile assumerà ad ogni iterazione un carattere appartenente alla stringa. Ad esempio:

```
stringa = 'LPO'
for c in stringa:
  print(c)
```

Stampa:

L

P

O

Classi Mixin

```
class CContaElementiMixin:
  def conta_elementi(self):
    num_elem = 0
    for elem in self.contenitore_di_elementi:
      num_elem = num_elem + 1
    return num_elem
```

Classi Mixin

```
class CStringa(CContaElementiMixin):
  def __init__(self, stringa):
    self.contenitore_di_elementi = stringa
  #... (supponiamo implementi metodi utili solo per le stringhe)

class CLista(CContaElementiMixin):
  def __init__(self, lista):
    self.contenitore_di_elementi = lista
  #... (supponiamo implementi metodi utili solo per le liste)
```

Classi Mixin

```
oggetto_stringa = CStringa("casa")
print(oggetto_stringa.conta_elementi())
```

Stamperà:

4

Esercizio sulle Classi Mixin

Supponete di voler aggiungere alle classi CStringa e CLista alcuni metodi che analizzano gli elementi duplicati. In particolare volete aggiungere i metodi:

- conta_duplicati, che restituisce il numero di caratteri / elementi della lista duplicati.
- cerca_duplicati, che restituisce una lista contenente i caratteri / gli elementi della lista duplicati.

Create quindi una mixin CAnalisiDuplicatiMixin.

Esercizio sulle Classi Mixin

```
class CStringa(CAnalisiDuplicatiMixin):
  def __init__(self, stringa):
    self.contenitore_di_elementi = stringa
  #... (supponiamo implementi metodi utili solo per le stringhe)

class CLista(CAnalisiDuplicatiMixin):
  def __init__(self, lista):
    self.contenitore_di_elementi = lista
  #... (supponiamo implementi metodi utili solo per le liste)
```

Esercizio sulle Classi Mixin

Vogliamo creare una lista di duplicati

duplicati = [].

Per fare questo, possiamo utilizzare una struttura dati di appoggio:

elementi_trovati = []

Scorriamo uno ad uno gli elementi del nostro contenitore di elementi.

Se l'elemento corrente non è presente in elementi_trovati, lo inseriamo in questa lista.

Se è già presente in elementi_trovati significa che è un duplicato. Lo inseriamo quindi nella lista duplicati.

Una volta che abbiamo esaminato tutti gli elementi dobbiamo semplicemente restituire la lista "duplicati".

Esercizio sulle Classi Mixin

```
class CAnalisiDuplicatiMixin:
  def cerca_duplicati(self):
    duplicati = []
    elementi_trovati = [] # struttura dati "di appoggio"
    for elem in self.contenitore_di_elementi:
      gia_trovato = False
      for elem_trovato in elementi_trovati:
        if elem == elem_trovato:
          duplicati.append(elem)
          gia_trovato = True
      if gia_trovato is False:
        elementi_trovati.append(elem)
    return duplicati
```

Esercizio sulle Classi Mixin

```
def conta_duplicati(self):
    duplicati = self.cerca_duplicati()
    return len(duplicati)
```

Esercizio sulle Classi Mixin

```
oggetto_stringa = CStringa("LPOOO")
print(oggetto_stringa.cerca_duplicati())
print(oggetto_stringa.conta_duplicati())
```

Stamperà:

['O', 'O']

2