Programmazione Avanzata e parallela

Lezione 21

Luca Manzoni

Python e Oggetti

Programmazione orientata agli oggetti (OOP)

- Copiare gli oggetti
- Ereditarietà
- Overriding dei metodi
- super()
- Ereditarietà multipla
- Ereditarietà vs composizione
- Eccezioni

Copia di oggetti

- Se lavoriamo su interi sappiamo che il seguente codice non modifica x:

```
x = 3
y = x
y = 4
```

- Cosa succede invece con gli oggetti?

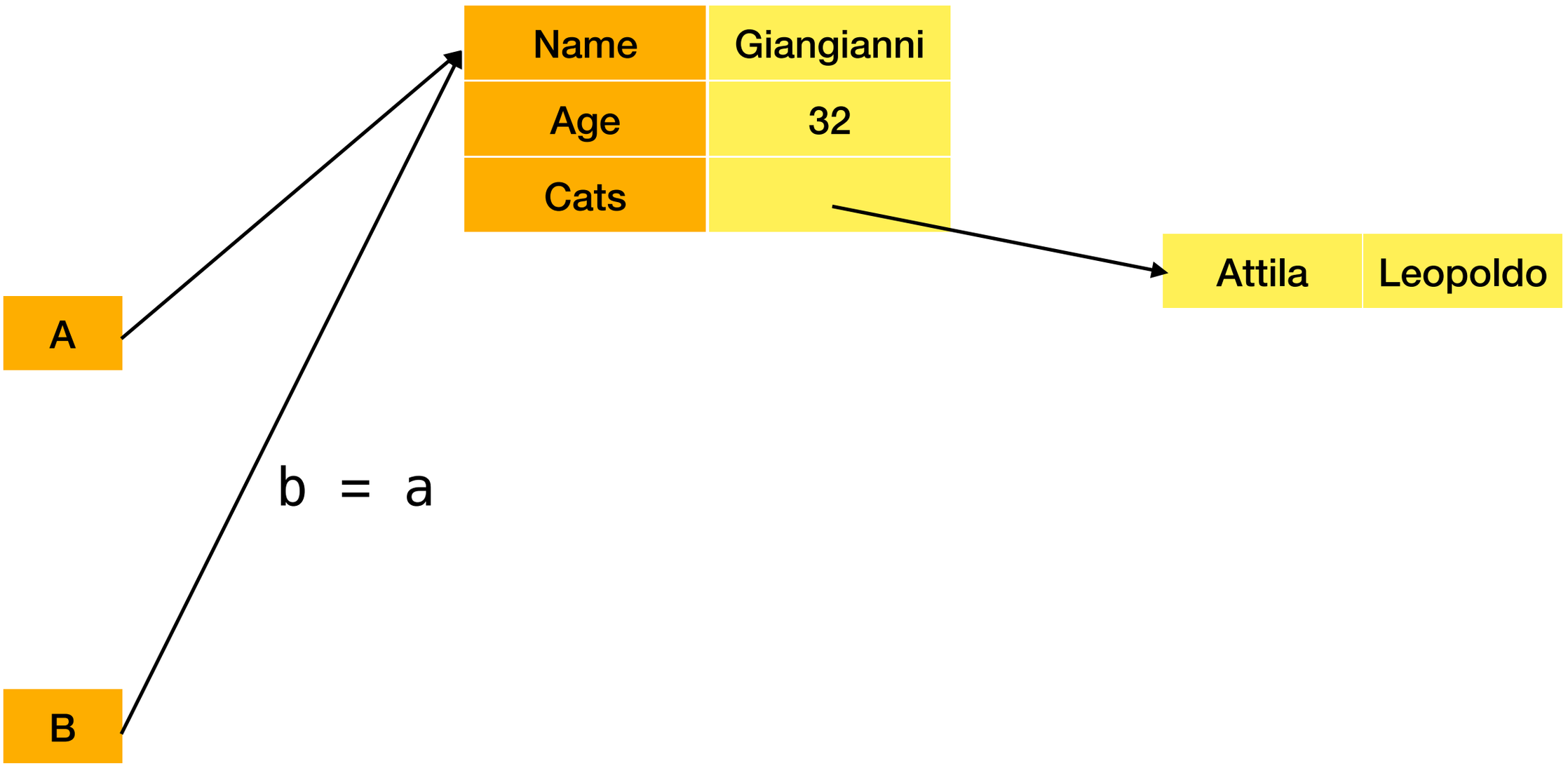
```
x = Person(“Mario”)
y = x
y.name = “Giovanni”
```

- Quando facciamo un assegnamento stiamo assegnando un riferimento…

- …e quindi lavoriamo sulla stessa istanza (i.e., c’è un solo oggetto, che non viene copiato)

Copia di oggetti

Copiare riferimenti



Copia di oggetti

Come risolvere?

- Possiamo pensare di avere un metodo statico per costruire una copia a partire da un originale

- Possiamo usare la libreria “copy”:

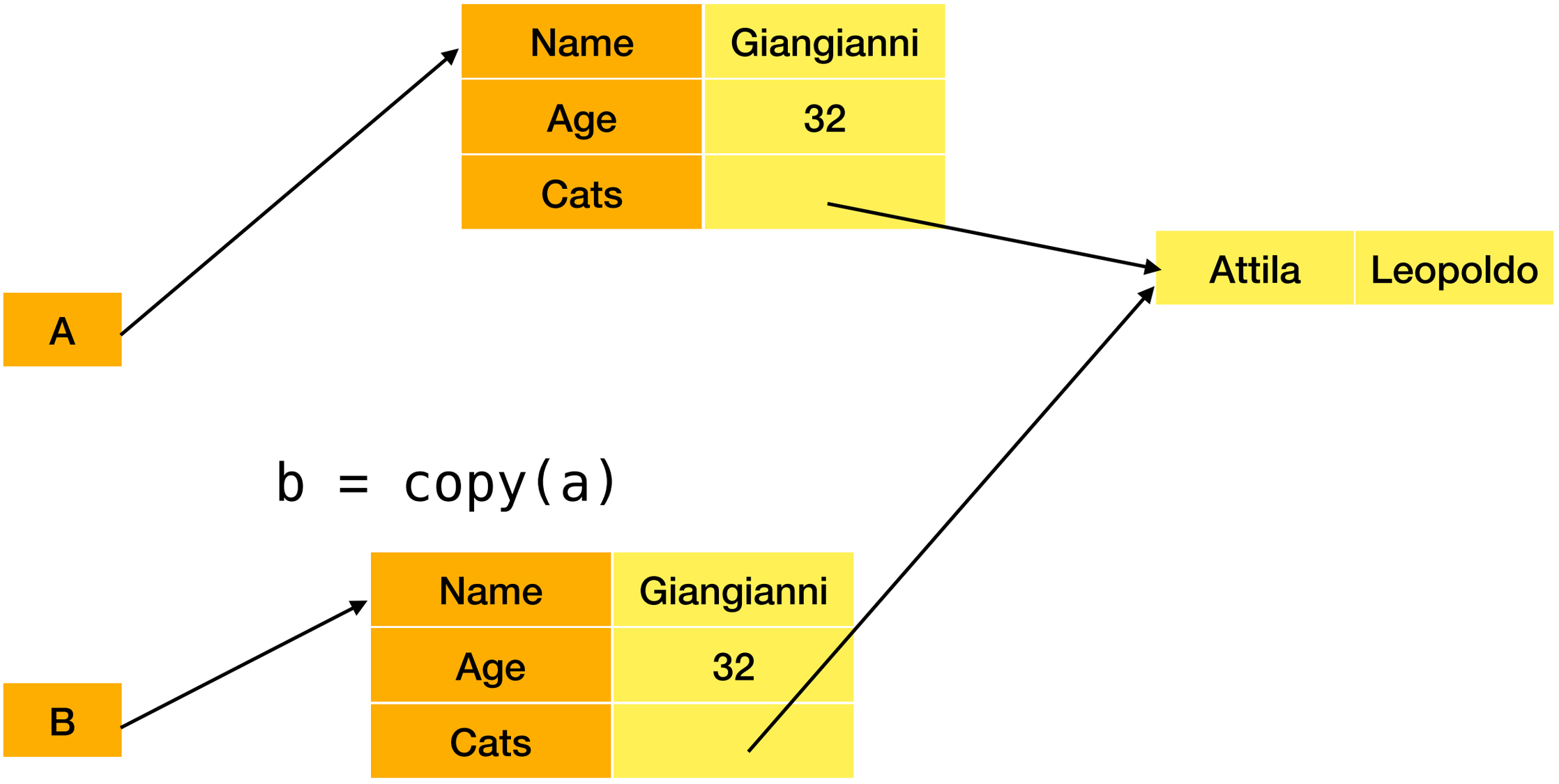
```
from copy import copy, deepcopy
```

- “*copy*” fornisce una copia “shallow” dell’oggetto: copia tutti gli attributi (facendo assegnamenti) in una nuova istanza dell’oggetto

- “*deepcopy*” fa invece una copia anche di tutti gli attributi, la differenza è si vede uno degli attributi è un oggetto!

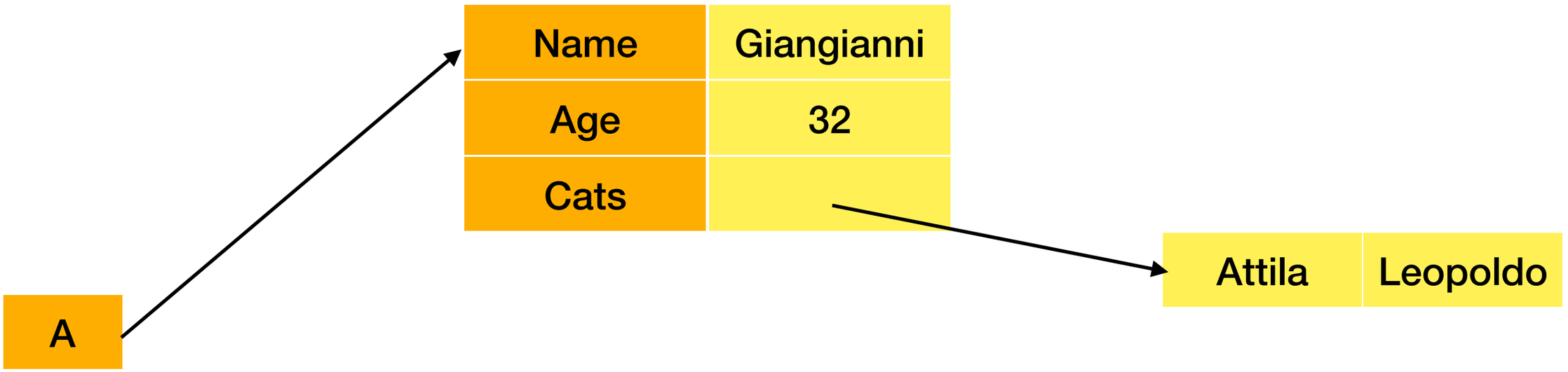
Copia di oggetti

Copy

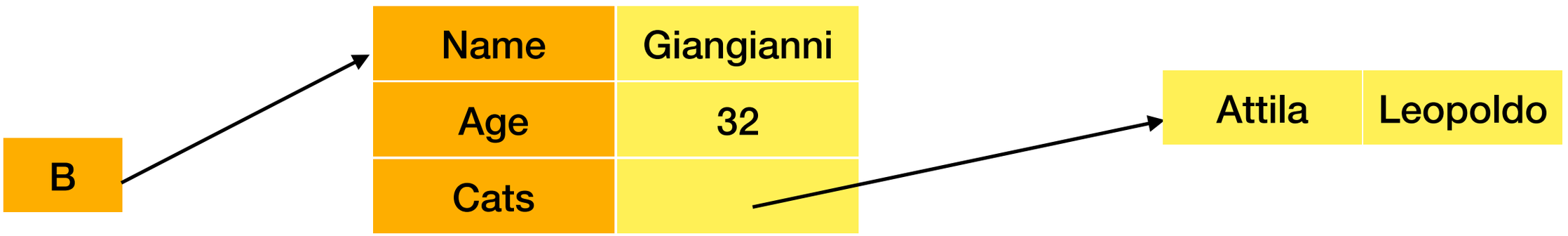


Copia di oggetti

Deepcopy




```
b = deepcopy(a)
```



Copia di oggetti

Come risolvere?

- Non sempre una copia shallow è un problema…
- …per esempio se comunque si deve lavorare su strutture condivise
- A volte serve fare operazioni speciali per la copia (e.g., se alcune risorse non devono o non possono essere copiate). In questo caso è possibile definire due metodi appositi:
  - `__copy__(self)`
  - `__deepcopy__(self)`

Ereditarietà

Motivazioni

- Spesso una serie di classi modellano concetti simili…
- …e quindi il contenuto dei metodi risulta praticamente duplicato
- Questo è dovuto al fatto che i metodi solo legati a una classe e quindi non possiamo dire direttamente “usa lo stesso metodo per entrambe queste classi”
- Potrebbe essere utile invece avere una classe più generica che contiene il codice comune…
- …e classi più specializzate che ereditano le parti comuni

Ereditarietà

Esempio
Motivazionale

```python
class Coniglio:

    common_name = "Coniglio"
    calories_needed = 200

    def __init__(self, name, age):
        self.name = name
        self.age = age
        self.happiness = 0
        self.calories = 0

    def eat(self, food):
        self.calories += food.calories
        if (self.calories > self.calories_needed):
            print("Ho mangiato troppo")
            self.happiness -= 1

    def interact_with(self, animal):
        self.happiness += 1
        print(f"Sto giocando con {animal.name}")

    def __str__(self):
        return f"Sono un {self.common_name} di nome {self.name}"
```

```python
class Gatto:

    common_name = "Gatto"
    calories_needed = 300

    def __init__(self, name, age):
        self.name = name
        self.age = age
        self.happiness = 0
        self.calories = 0

    def eat(self, food):
        self.calories += food.calories
        if (self.calories > self.calories_needed):
            print("Ho mangiato troppo")
            self.happiness -= 1

    def interact_with(self, animal):
        self.happiness += 1
        print(f"Sto giocando con {animal.name}")

    def __str__(self):
        return f"Sono un {self.common_name} di nome {self.name}"
```

Molti dei metodi sono duplicati!

Ereditarietà

Motivazioni

- Possiamo pensare a una classe comune (detta “base class” o “superclass”) di tipo Animale
- Questa classe includerà il codice comune
- SI creeranno due classi derivate (“Gatto” e “Coniglio”) che **ereditano** da Animale:
  - Dove non specificato diversamente si comporteranno come la classe base
  - Definiscono solo i comportamenti in cui differiscono (attributi e metodi)

Ereditarietà

Come definirla

- Dire che una classe derivata eredita da una classe base è indicato nel seguente modo:

Stiamo indicando che “Derived” è una sottoclasse di “Base”

```
class Derived(Base):
```

Ereditarietà

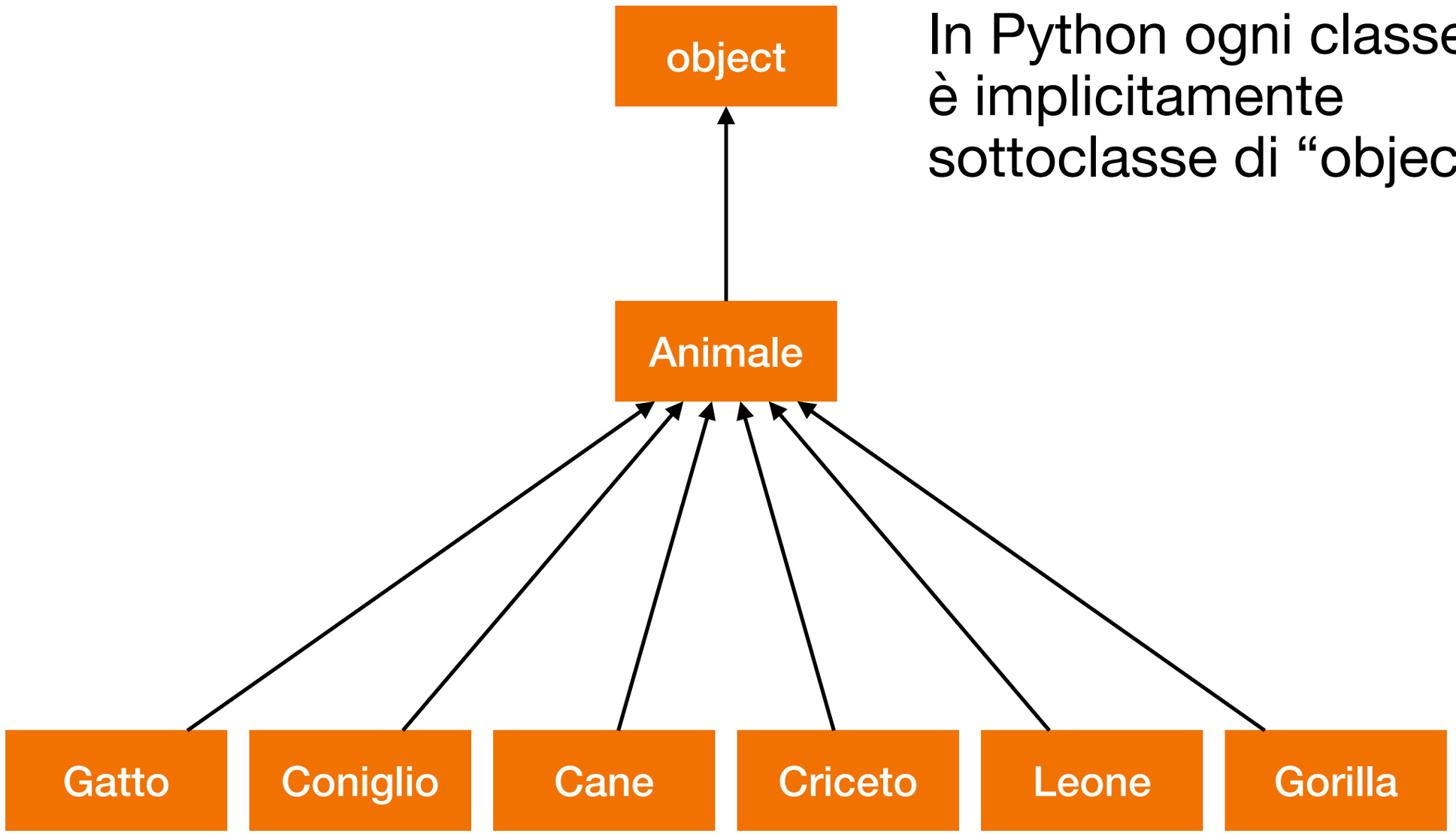
Casi semplici

- Il caso più semplice è una classe derivata che si comporta esattamente come la classe base:
  - `class BlobSenzaForma(Animale):`
    `pass`
- Un caso più interessante è la definizione di nuovi metodi o nuovi attributi:
  - `class Gatto(Animale):`

    `def conquista_il_mondo(self):`
    `…`

Ereditarietà

Gerarchia delle classi



In Python ogni classe è implicitamente sottoclasse di “object”

Overriding dei metodi

Ridefinire i metodi

- Possiamo anche definire dei metodi che erano già stati definiti nella classe base
- Questo non è un problema (si dice che facciamo **overriding** del metodo), verrà invocato il metodo ridefinito invece di quello della classe base
- Questo sostituisce il metodo della classe base che non verrà più chiamato
- E se volessimo chiamare esplicitamente un metodo della superclasse?
- Questo è utile, per esempio, se volessimo cambiare __init__

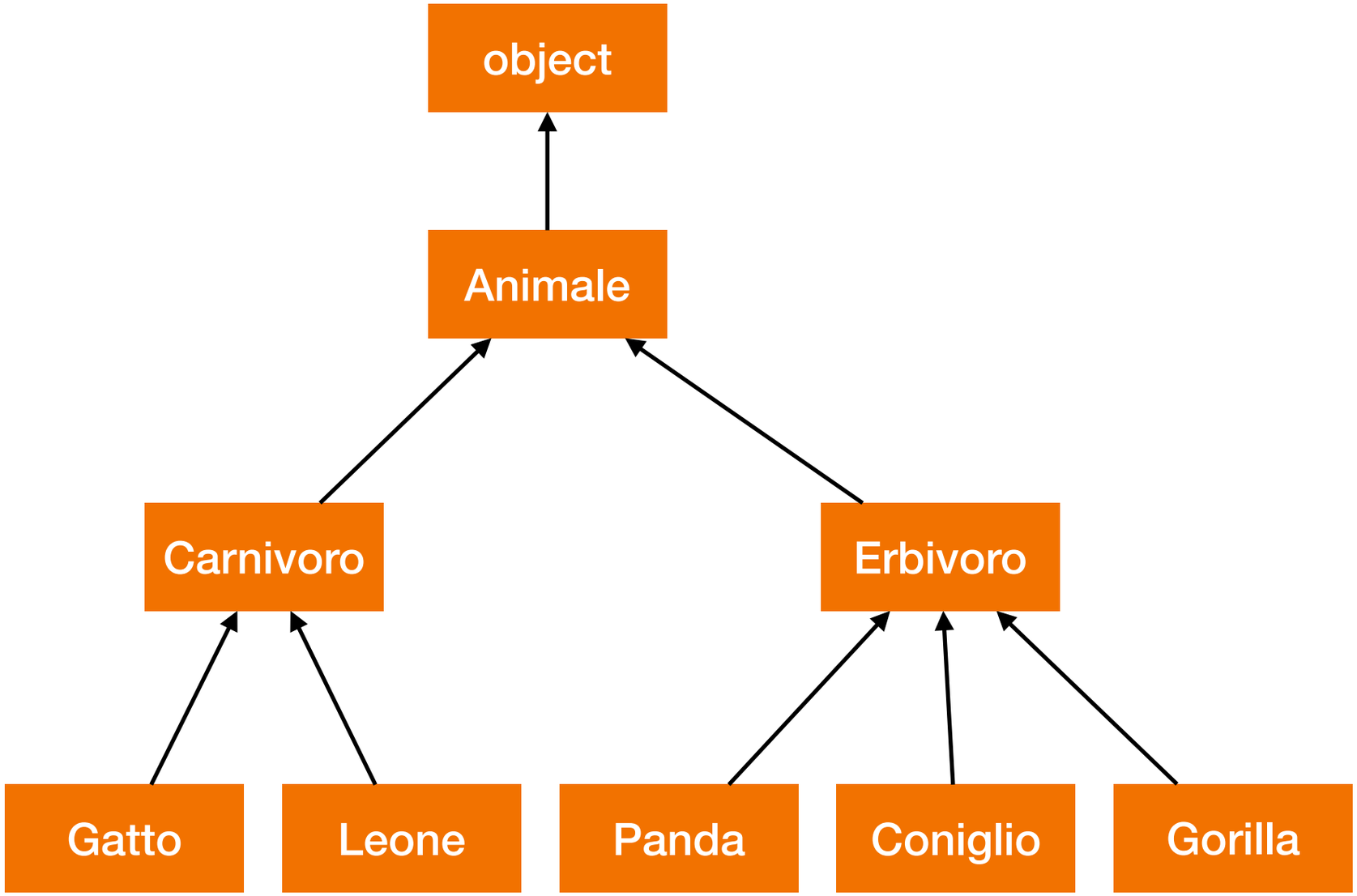
super()

Accedere alla superclasse

- Possiamo usare "*super()*" per trovare la superclasse della classe attuale e chiamare dei metodi su di essa
- E.g., stiamo ridefinendo *__init__* ma vogliamo chiamare anche il metodo *__init__* della superclasse
- Generalmente sappiamo chi sia la superclasse…
- …ma è buona norma usare "*super()*" per riferirsi alla superclasse
- In questo modo non è "hard-coded" in molti punti del codice

Ereditarietà

Una gerarchia più complessa



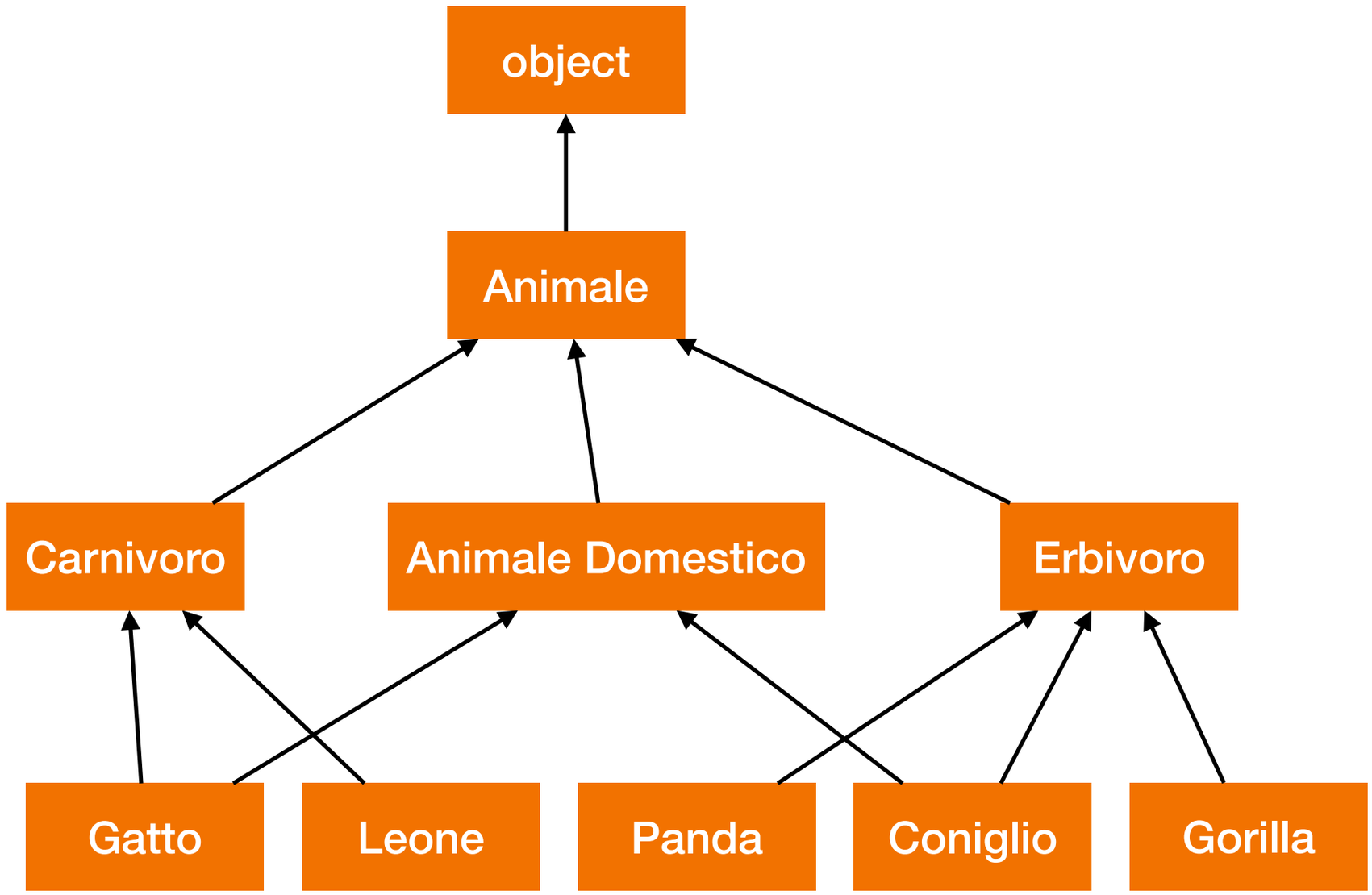
Ereditarietà multipla

Ereditare da più classi

- Una classe può ereditare da più di una singola classe base
- La classe derivata può accedere agli attributi di tutte le classi da cui eredita
- Quando chiamiamo un metodo definito in una sola della classi base è chiaro quale venga chiamato…
- …ma se lo definiscono più classi base?
- Possiamo vedere l'ordine seguito per l'ordine con cui viene cercato il metodo usando
  NomeClasse.mro() (Method Resolution Order)

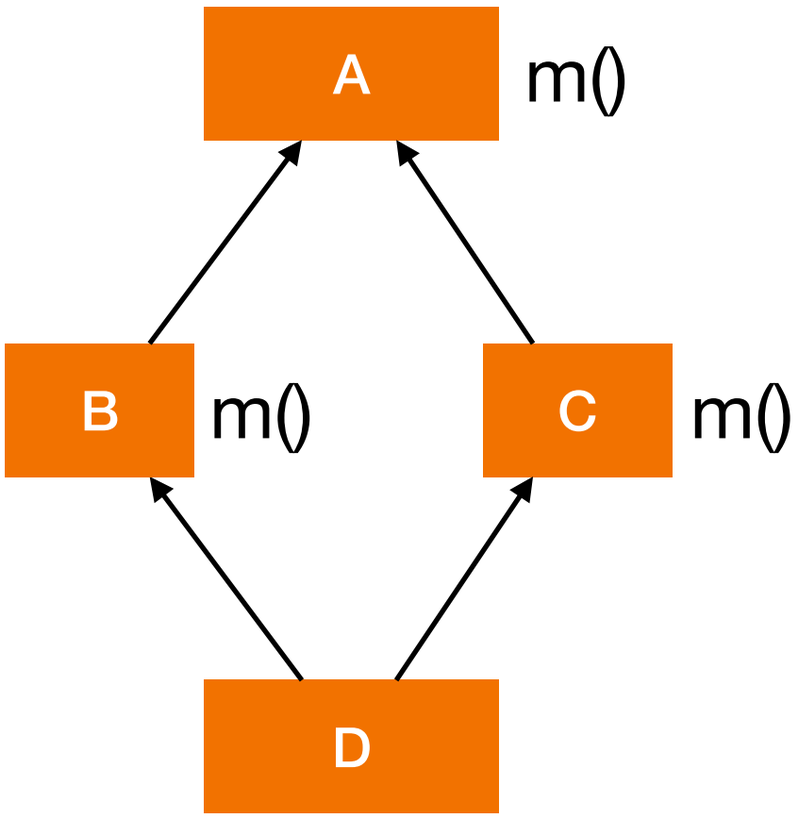
Ereditarietà

Con ereditarietà multipla



Ereditarietà

Il “diamond problem”



Se chiamo il metodo m() in D quale è il metodo invocato?

In python dipende dall’ordine in cui ereditiamo da B e C:

D(B,C) chiamerà m() di B

D(C,B) chiamerà m() di C

Ereditarietà multipla

Ereditarietà singola e interfacce

- Ereditare da più classi può permetterci di semplificare il codice
- In molti linguaggi è consentita solo ereditarietà singola (si eredita da una sola classe)…
- …ma è fornita la possibilità di implementare interfacce, ovvero di dire che una classe A deve implementare un certo insieme di metodi B
- In questo modo sappiamo che una classe A che implementa l'interfaccia B mette a disposizione tutti i metodi dichiarati in B

Ereditarietà vs composizione

Quando usare cosa

- L'**ereditarietà** funziona meglio per le relazioni "**is-a**"
  - Un coniglio è un animale
  - Quindi la classe "coniglio" eredita dalla classe "animale"
- La **composizione** funziona meglio per le relazioni di tipo "**has-a**"
  - Uno riserva naturale ha degli animali che la abitano
  - Quindi ha senso che la classe "riserva naturale" abbia come attributo una lista di animali

Eccezioni

E loro gestione

- In Python (come in molti linguaggi moderni) possiamo usare le **eccezioni**
- Quando viene lanciata una eccezione si risale lo stack delle chiamate fino a trovare chi può gestirla
- Abbiamo quindi due aspetti essenziali:
  - Come si *lancia una eccezione* per unificare una condizione eccezionale (errore in lettura, chiave mancante)
  - Come si *cattura una eccezione* per decidere come gestirla

Eccezioni

Come catturarle

- Possiamo gestire le eccezioni come segue:

Stiamo indicando che il seguente codice può generare una eccezione

```
try:
    codice_che_può_lanciare_eccezione
except ClasseDellEccezione:
    gestione_eccezione
```

Indichiamo che il codice che segue è da eseguire se incontriamo una eccezione di tipo “ClasseDellEccezione”

Eccezioni

E loro gestione

- “Risaliamo” lo stack fino a trovare un blocco try…except in grado di gestire l’eccezione
- Possiamo avere più clausole “except” con classi diverse
- Verrà usato il codice nella prima che fa match
- Se facciamo “except A” e B è sottoclasse di A allora abbiamo un match…
- Quindi usiamo le clausole except sempre dalla classe più specifica a quella più generica

Eccezioni

Come lanciarle/sollevarle

- Possiamo sollevare una eccezione come segue:

Stiamo indicando che l'oggetto che segue indica l'eccezione generata, tipi diversi di oggetto per eccezioni differenti

```
raise ClasseEccezione()
```

Possiamo definire noi una classe specifica (magari ereditando da "Exception") o usare delle classi predefinite

Eccezioni

Alcune note

- La classe *Exception* ha un metodo "*add_note(note)*" che serve per aggiungere informazioni all'eccezione (e.g., un file non può essere aperto e indichiamo il nome del file, etc)

- Possiamo rilanciare l'eccezione all'interno di "except" chiamando **raise** (senza nulla dopo)

- Possiamo dare un nome all'eccezione catturata:
  except ClasseEccezione as nome:

- Possiamo aggiungere una clausola "**finally:**" dopo i vari "**except**" per indicare del codice da eseguire sempre (indipendentemente dal fatto che l'eccezione ci sia stata o no)