Object-Oriented Programming in Python

Advanced Computer Programming

Prof. Luigi De Simone

Sommario

- Classi in Python
- *Getters* and *setters*
- *Information hiding*
- Ereditarietà
- Polimorfismo
- Classi astratte e Interfacce

Riferimenti

- Tony Gaddis. **Introduzione a Python**. 5° ed. – Pearson, 2021
- Paul J. Deitel, Harvey M. Deitel. **Introduzione a Python. Per l'informatica e la data science.** Pearson, 2021
- **Python: How to Think Like a Computer Scientist interactive edition** https://runestone.academy/runestone/books/published/thinkcspy/index.html
- Allen Downey. **Think Python** - https://greenteapress.com/thinkpython2/thinkpython2.pdf

Classi in Python

Oggetti in Python

- Python supporta diversi tipi di dato

```
1234        3.14159        "Hello"        [1, 5, 7, 11, 13]

{"CA": "California", "MA": "Massachusetts"}
```

- Ognuno è un **oggetto istanza di una classe**, e ogni oggetto ha:
  - Un **tipo (la classe)**
  - Un **rappresentazione dei dati** interna (tipo di dato primitivo o strutturato)
  - Un insieme di procederue per l'**interazione** con gli oggetti
- Un oggetto è un'**istanza** di un tipo
  - `1234` è istanza di un `int`
  - `"hello"` è istanza di una stringa

Python e OOP

- Possiamo **creare nuovi oggetti** di qualche tipo
- Possiamo **manipolare oggetti**
- Possiamo **distruggere oggetti**
  - **Esplicitamente** usando `del` oppure "dimenticandoci" di loro
  - Python effettuarà un *reclaim* degli oggetti distrutti o inaccessibili attraverso un sistema di "**garbage collection**"

Ricordiamo cos'è un oggetto

- Gli oggetti sono **astrazioni di dati** che includono

(1)una **rappresentazione interna**

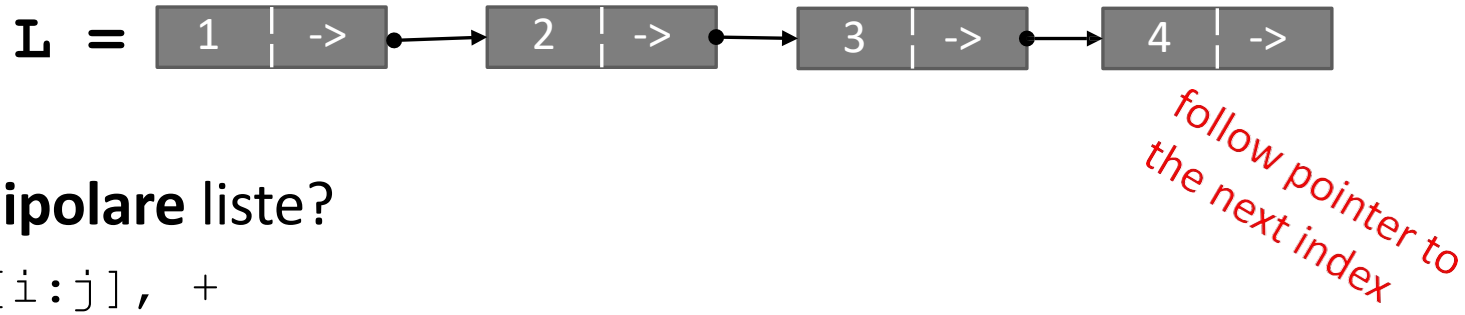
- Attraverso **attributi dati membro**

(2)una **interfaccia** per l'interazione con l'oggetto

- Attraverso i **metodi membro**
- Definisce i comportamenti ma nasconde l'implementazione (*information hiding*)

Esempio: Il tipo `list`

- Come sono **rappresentate internamente** le liste? Lista *linked* di celle di memoria



- Come **manipolare** liste?
  - `L[i], L[i:j], +`
  - `len(), min(), max(), del(L[i])`
  - `L.append(),L.extend(),L.count(),L.index(),`
    `L.insert(),L.pop(),L.remove(),L.reverse(), L.sort()`
- La **rappresentazione interna** dovrebbe essere **privata**
- Il corretto comportamento di una lista può essere compromesso **se manipoliamo la rappresentazione interna direttamente**

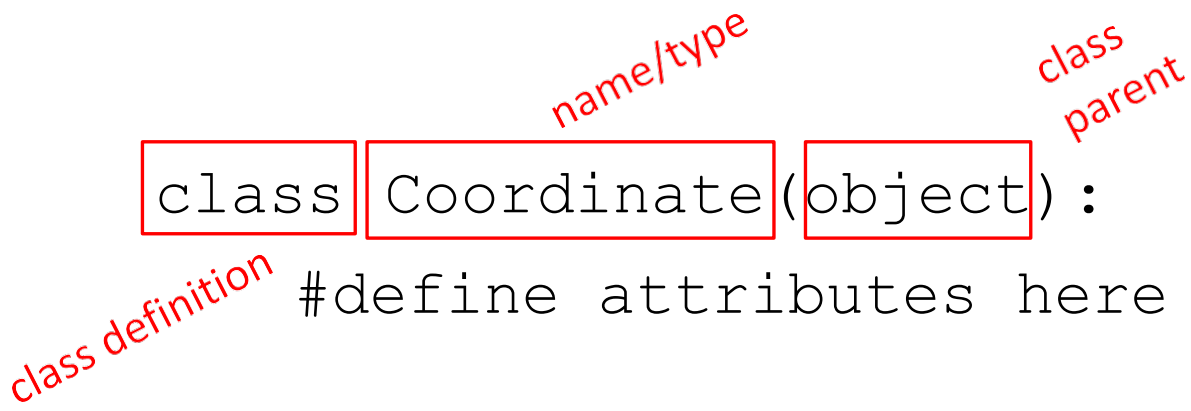
Uso delle classi in Python

- Facciamo distinzione tra **creare una classe** e **utilizzare una istanza (oggetto)** di una classe

- **Creare** la classe prevede di
  - Definire il **nome** della classe
  - Definire gli **attributi** di una classe
- **Utilizzare** la classe include
  - Creare una nuova **instanza di una classe** (oggetto)
  - Effettuare delle **operazioni con l'istanza**

Implementazione di una classe

- Uso della keyword `class` per definire un nuovo tipo



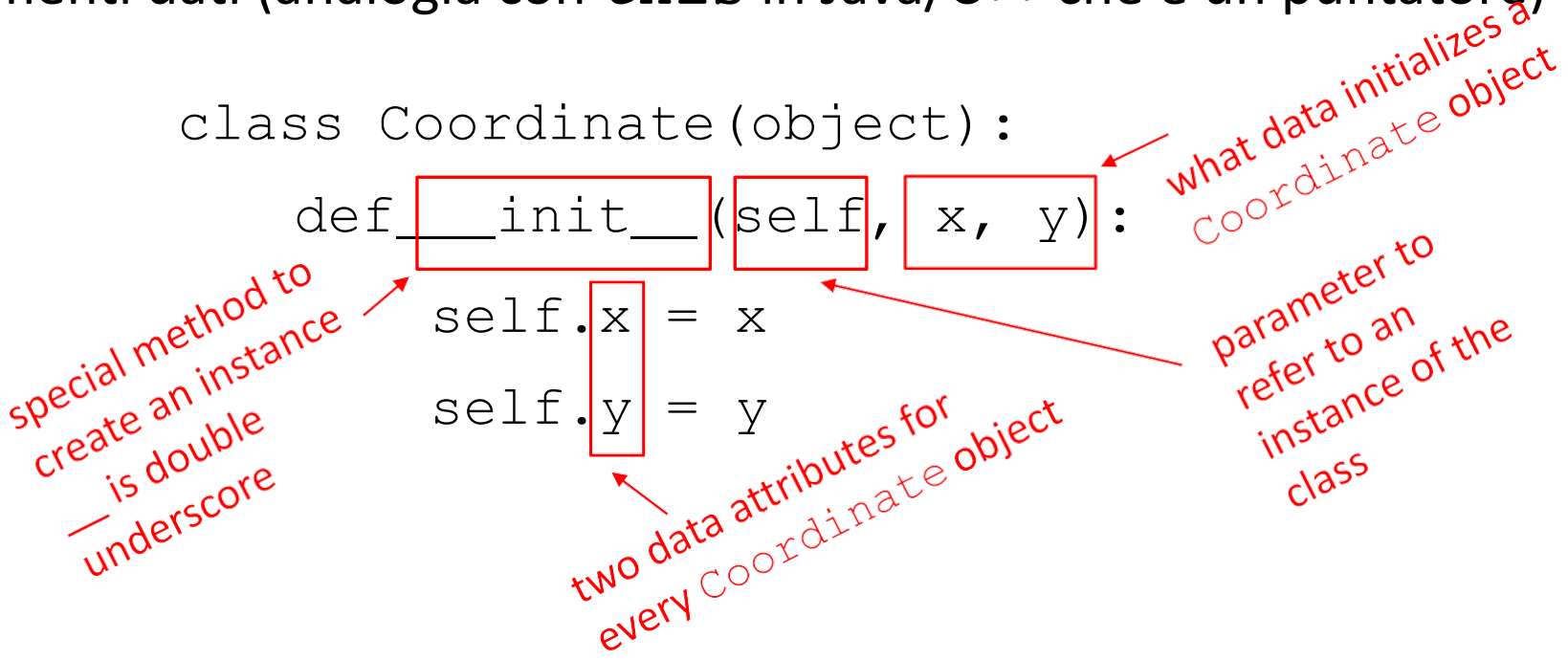
- Simile a `def`, l'indentazione del codice indica che le istruzioni fanno parte della **definizione di classe**
- La parola `object` significa che `Coordinate` è un oggetto Python ed **eredita** tutti i suoi attributi
  - `Coordinate` è una sottoclasse di `object`
  - `object` è una superclasse (classe madre) di `Coordinate`

Cosa sono gli attributi?

- Dati e procedure che “**appartengono**” alla classe
- **Attributi dati**
  - Pensare ai dati come altri oggetti che possono costruire la classe
  - *E.g., una coordinata è fatta da numeri*
- **Metodi** (*attributi procedurali*)
  - Pensare ai metodi come funzioni che possono essere utilizzate solo nel contesto della classe
  - Permettono di interagire con l’oggetto istanza della classe definita
  - E.g., *possiamo definire la distanza tra 2 oggetti coordinate, ma non c’è significato di distanza tra 2 oggetti lista nel contesto della classe Coordinate*

Come creare un'istanza di una classe

- Dobbiamo definire come creare un'istanza di un oggetto, il **costruttore**
- Utilizzare un metodo speciale chiamato **`__init__`** per inizializzare (alcuni) dati attributi
- La parola chiave **`self`** rappresenta l'istanza della classe e lega gli attributi con gli argomenti dati (analogia con `this` in Java/C++ che è un puntatore)



Istanza di classe

```
c = Coordinate(3,4)
origin = Coordinate(0,0)
print(c.x)
print(origin.x)
```

create a new object of type `Coordinate` and pass in 3 and 4 to the `__init__`

use the dot to access an attribute of instance `c`

- I dati attributi di un'istanza sono chiamati **variabili di istanza**
- Se non forniamo un argomento per `self`, Python utilizzerà un argomento di default

Metodi di classe

- Sono attributi procedurali, come se fossero **funzioni che possono essere utilizzate solo nel contesto di tale classe**
- Python di default passa sempre **un oggetto come primo parametro alla chiamata di un metodo**
  - Per convenzione si utilizza `self` come nome del primo parametro di tutti i metodi di una classe
- L'**operatore** "**.**" è utilizzato per accedere gli attributi di una classe
  - Un attributo dato di un oggetto
  - Un metodo di un oggetto

Esempio: Definire un metodo per la classe `Coordinate`

```
class  Coordinate(object):
    def_init__(self, x, y):
        self.x = x
        self.y = y
    def distance(self, other):
        x_diff_sq = (self.x-other.x)**2
        y_diff_sq = (self.y-other.y)**2
        return (x_diff_sq + y_diff_sq)**0.5
```

use it to refer to any instance

another parameter to method

dot notation to access data

- Eccetto `self` e la notazione punto, i metodi di una classe si comportano semplicemente come funzioni

Come utilizzare un metodo

```
def distance(self, other):
    # code here
```

method def

Utilizzare la classe:

- **Modo convenzionale**

```
c = Coordinate(3,4)
zero = Coordinate(0,0)
print(c.distance(zero))
```

object to call method on

name of method

Parameter not including `self`, (`self` is implied to be c)

- **Modo equivalente**

```
c = Coordinate(3,4)
zero = Coordinate(0,0)
print(Coordinate.distance(c, zero))
```

name of class

name of method

parameters, including an object to call the method on, representing `self`

Il metodo `__str__`

```
>>> c = Coordinate(3,4)
>>> print(c)
<__main__.Coordinate object at 0x7fa918510488>
```

- Di default la `print` di un oggetto restituisce una rappresentazione **non informative**
- Dobbiamo definire il **metodo `__str__`** per una classe se vogliamo fornire info diverse
- Python chiama **`__str__`** quando invochiamo la print su un oggetto istanza di una classe
- Assumiamo che per un oggetto di tipo `Coordinate` vogliamo mostrare il seguente:

```
>>> print(c)
<3,4>
```

Esempio: Il metodo `__str__`

```
class Coordinate(object):
    def_init__(self, x, y):
        self.x = x
        self.y = y
    def distance(self, other):
        x_diff_sq = (self.x-other.x)**2
        y_diff_sq = (self.y-other.y)**2
        return (x_diff_sq + y_diff_sq)**0.5
    def __str__(s elf):
        return "<"+str(self.x)+","+str(self.y)+">"
```

name of special method

must return a string

Tipi e Classi. `isinstance()`

- Possiamo ottenere il tipo di un oggetto utilizzando il metodo **`isinstance()`**

```
>>> c = Coordinate(3,4)
>>> print(c)
<3,4>
>>> print(type(c))
<class___main__.Coordinate>
```

return of the __str__ method

the type of object c is a class Coordinate

- Ha senso perché:

```
>>> print(Coordinate)
<class___main__.Coordinate>
>>> print(type(Coordinate))
<type 'type'>
```

a Coordinate is a class

a Coordinate class is a type of object

- Utilizzare **`isinstance()`** per controllare se un oggetto è di un certo tipo target

```
>>>print(isinstance(c,Coordinate))
True
```

Operatori speciali

- `+, -, ==, <, >, len(), print,` e molti altri (ref. https://docs.python.org/3/reference/datamodel.html#basic-customization)
- Come `print`, possiamo fare l'*override* di questi operatori nel contesto della definizione di classi
- Definirli con il *doppio underscore* prima e dopo

```
__add__(self, other)    →    self + other
__sub__(self, other)    →    self - other
__eq__(self, other)     →    self == other
__lt__(self, other)     →    self < other
__len__(self)           →    len(self)
__str__(self)           →    print self
```

... e altri

Esempio: La classe `Fraction`

- Creare un nuovo tipo per rappresentare un numero come **frazione**
- **La rappresentazione interna** prevede **due interi**
  - Numeratore
  - Denominatore
- **Metodi**
  - Addizione, sottrazione
  - Stampa della rappresentazione, conversione a float
  - Inversione di una frazione

Definizione e istanza di classe

Definizione di una classe

- Il nome della classe è il **tipo**

```
class Coordinate(object)
```

- Usare `self` per riferirsi alle istanze di classe

```
(self.x - self.y)**2
```

  - `self` è il parametro utilizzato dai metodi membri nella definizione di classe

- `class` definisce i dati e metodi membro comuni a tutte le istanze di classe

Istanza di una classe

- L'oggetto è una specifica istanza di classe

```
coord = Coordinate(1,2)
```

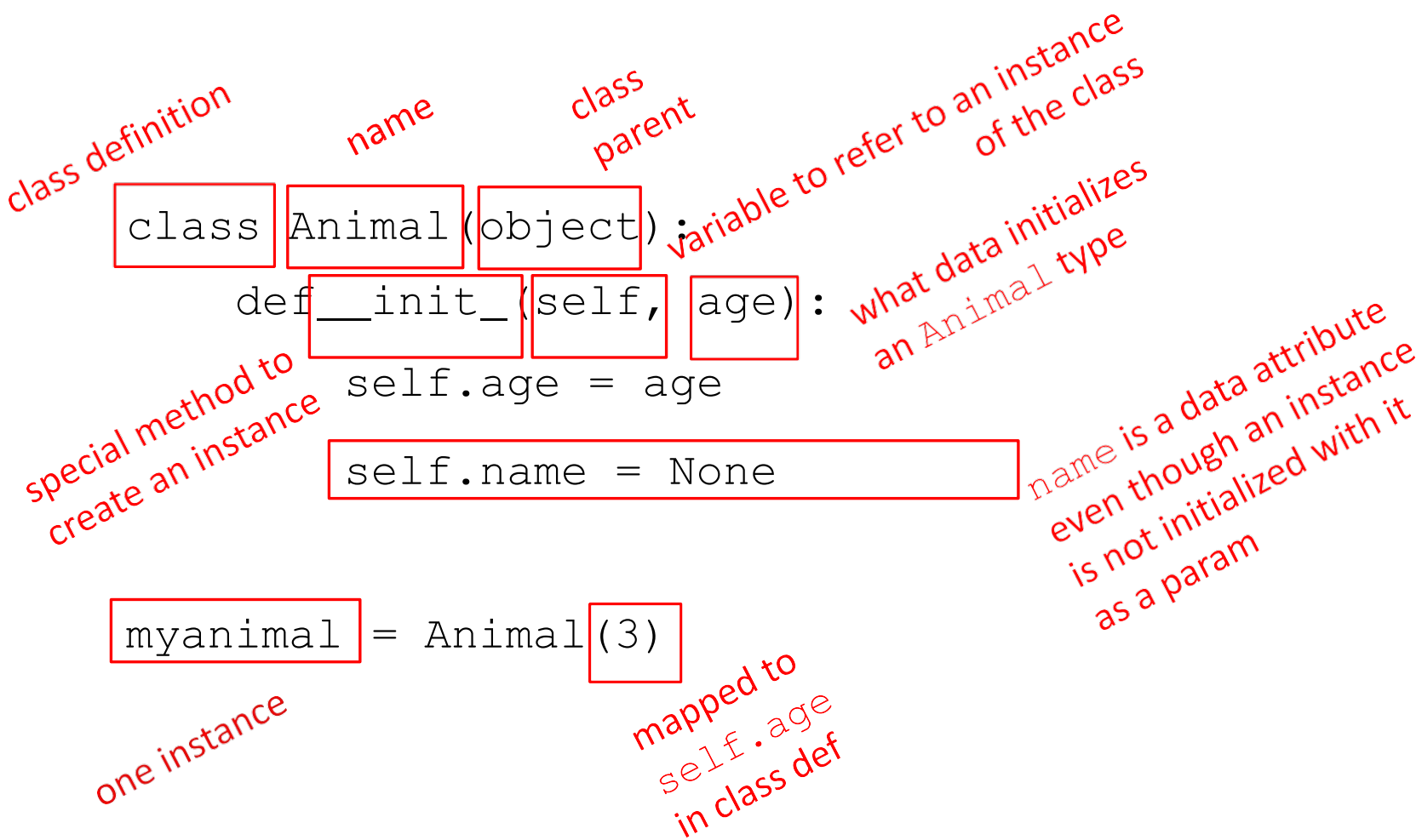
- I dati membro possono variare per ogni istanza

```
c1 = Coordinate(1,2)
c2 = Coordinate(3,4)
```

  - `c1` e `c2` hanno differenti valori per i dati membro

- L'istanza di una classe possiede la **struttura della classe**

Definire una classe



Metodi *getter* e *setter*

```
class Animal(object):
    def_init_(self, age):
        self.age = age
        self.name = None
    def get_age(self):
        return self.age
    def get_name(self):
        return self.name
    def set_age(self, newage):
        self.age = newage
    def set_name(self, newname=""):
        self.name = newname
    def__str__(self):
        return "animal:"+str(self.name)+":"+str(self.age)
```

getter

setter

- **getters e setters** devono essere utilizzati al di fuori della classe per accedere ai dati membro (*information hiding*)

Information hiding in Python

- Nella definizione di classe potremmo **modificare i dati membro** cambiando il nome

replaced `age` data attribute by `years`

```
class Animal(object):
    def___init__(self, age):
        self.years = age
    def get_age(self):
        return self.years
```

- Se proviamo ad **accedere ai dati membro** al di fuori della classe, e la definizione per i dati membro cambia potremmo incorrere in errori!

- All'esterno della classe è buona norma usare i metodi *getters* and *setters* *(e.g.,* usare `a.get_age()` e non `a.age`)
  - Buono stile di programmazione
  - Facilita la manutenzione del codice
  - Evita i bug!

Information hiding in Python

Python non è un buon linguaggio per l'information hiding!

- Consente di **accedere ai dati** dall'esterno di una definizione di classe

```
print(a.age)
```

- Consente di **scrivere i dati** dall'esterno di una definizione di classe

```
a.age = 'infinite'
```

- Consente di **creare dati membro** per un oggetto dall'esterno di una definizione di classe

```
a.size = "tiny"
```

- Nessuno di questi esempio rientra nei **buoni modi di programmare considerando l'*information hiding***

Argormenti di default

- **Gli argormenti di default** per i *parametri formali* sono utilizzati se nessun *parametro attuale* viene fornito

```
def set_name(self, newname=""):
    self.name = newname
```

- Per esempio, utilizzando i parametri di default

```
a = Animal(3)
a.set_name()
print(a.get_name())
```

prints ""

- non utilizzando i parametri di default

```
a = Animal(3)
a.set_name("fluffy")
print(a.get_name())
```

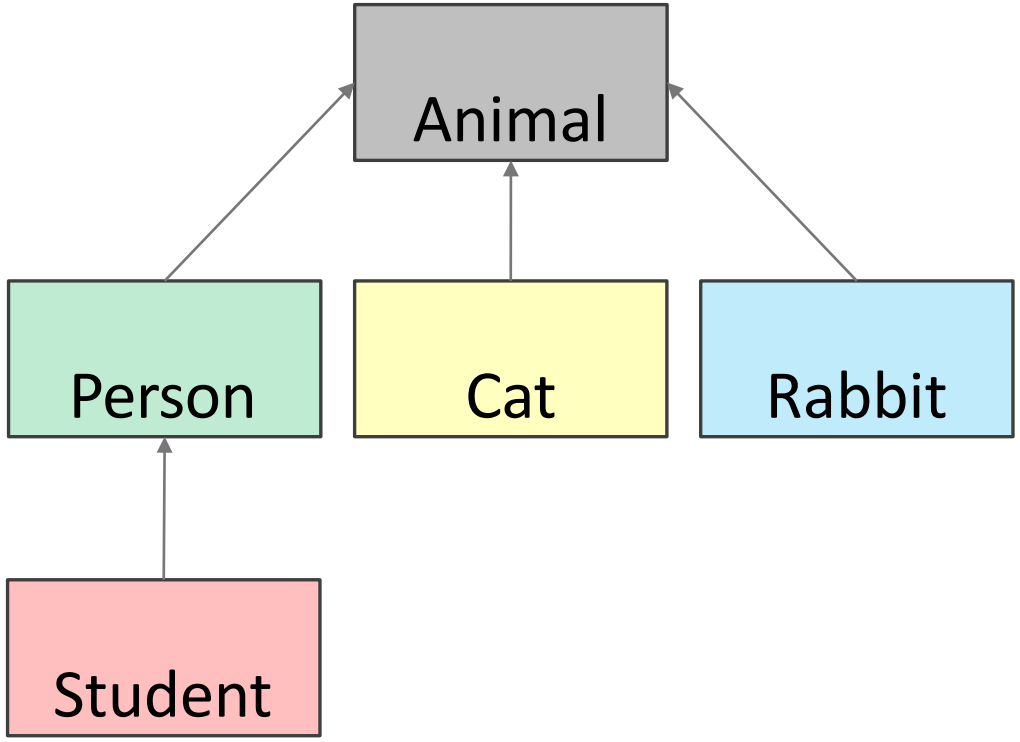
prints "fluffy"

Ereditarietà in Python:

Ereditarietà multipla, Overload degli operatori, Definizione di eccezioni

Ereditarietà in Python

- **Class padre** (superclass)
- **Classe figlia** (subclass)
  - **eredita** tutti i dati e metodi membro della classe padre
  - **aggiunge** più **info**
  - **aggiunge** più **comportamenti**
  - **sovrascrive** comportamenti
- Python supporta anche l'**ereditarietà multipla**

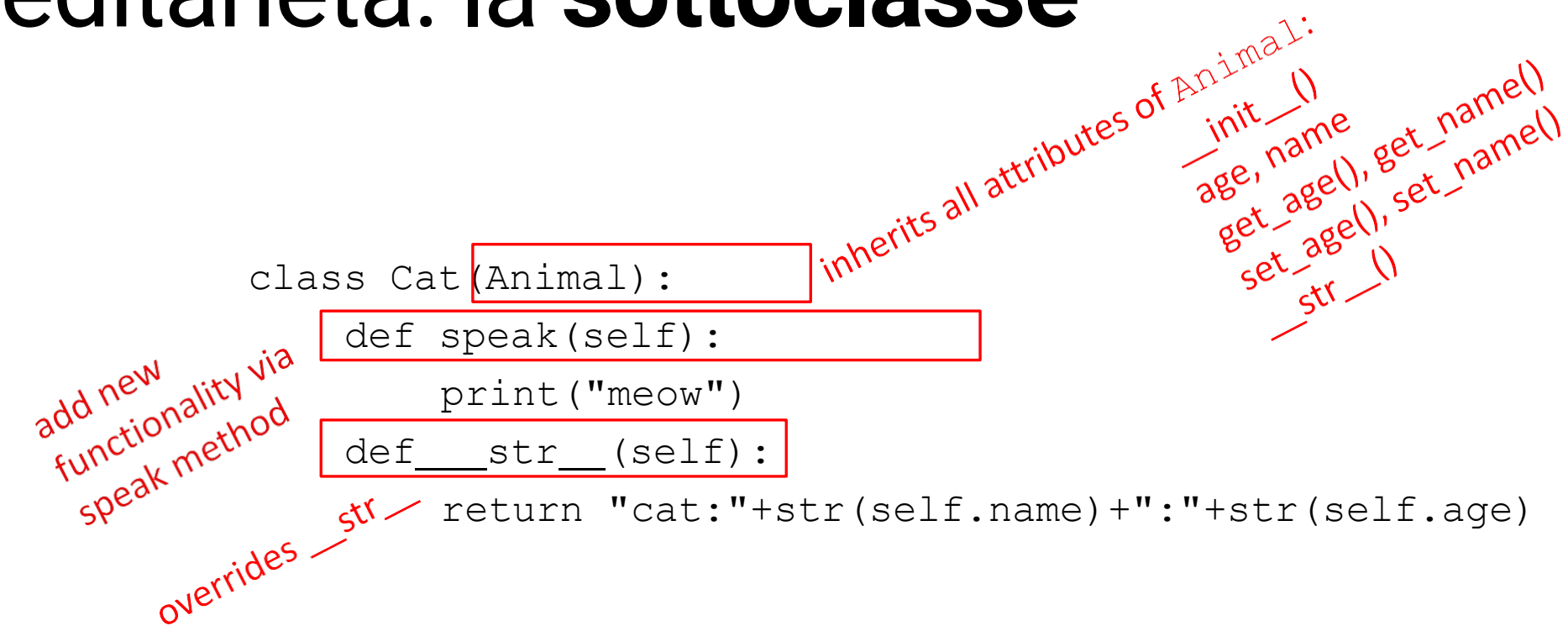


Ereditarietà: la **superclasse**

```
class Animal(object):
    def_init_(self, age):
        self.age = age
        self.name = None
    def get_age(self):
        return self.age
    def get_name(self):
        return self.name
    def set_age(self, newage):
        self.age = newage
    def set_name(self, newname=""):
        self.name = newname
    def___str__(self):
        return "animal:"+str(self.name)+":"+str(self.age)
```

- La classe Animal **eredità dalla classe base `object`**
- La classe `object` implementa tutte **le operazioni di base** in Python, come il binding di variabili, etc.

Ereditarietà: la **sottoclasse**



- **Aggiunge nuove funzionalità** con il metodo **`speak()`**
  - Su oggetti di tipo `Cat` posso invocare i nuovi metodi
  - Su oggetti di tipo `Animal` saranno sollevate eccezioni se provo ad invocare il nuovo metodo definito in `Cat`
- **Il metodo `__init__` non è mancante, semplicemente riuso quello fornito nella classe padre** (`Animal`)

Invocazione dei metodi in gerarchie di classi

- Una sottoclasse può avere dei **metodi con lo stesso nome** di quelli forniti dalla superclasse
  - Per un'istanza di un classe si cerca dapprima il nome di metodo nella definizione di classe corrente
  - Se il metodo non viene trovato, si cerca il metodo **risalendo la gerarchia**
  - Si usa il primo metodo trovato risalendo la gerarchia

Esempio di gerarchia

```
class Person(Animal):
    def __init__(self, name, age):
        Animal.__init__(self, age)
        self.set_name(name)
        self.friends = []
    def get_friends(self):
        return self.friends
    def add_friend(self, fname):
        if fname not in self.friends:
            self.friends.append(fname)
    def speak(self):
        print("hello")
    def age_diff(self, other):
        diff = self.age - other.age
        print(abs(diff), "year difference")
    def __str__(self):
        return "person:"+str(self.name)+":"+str(self.age)
```

La classe padre di `Person` è `Animal`

Chiama **esplicitamente** il costruttore di `Animal`

Chiama il metodo di `Animal` e aggiunge un nuovo dato membro

Nuovi metodi

Override del metodo `__str__` della classe `Animal`

```
import random
```

Importa metodi dalla classe `random`

```
class Student(Person):
    def __init__(self, name, age, major=None):
        Person.__init__(self, name, age)
        self.major = major
    def change_major(self, major):
        self.major = major
    def speak(self):
        r = random.random()
        if r < 0.25:
            print("i have homework")
        elif 0.25 <= r < 0.5:
            print("i need sleep")
        elif 0.5 <= r < 0.75:
            print("i should eat")
        else:
            print("i am watching tv")
    def__str_(self):
        return
        "student:"+str(self.name)+":"+str(self.age)+":"+str(self.major)
```

Eredita dati e metodi dalle classi `Person` e `Animal`

Aggiunge nuovi dati (nome dell'esame)

Il metodo random() restituisce un float in [0.0, 1.0)

Ereditarietà multipla: **esempio**

```
class Base1:
        Body of the class

class Base2:
     Body of the class

class Derived(Base1, Base2):
     Body of the class
```

Ereditarietà multipla

- Abbiamo parlato di ereditarietà, o più specificamente di "ereditarietà singola". Come abbiamo visto, in questo caso una classe eredita da una sola classe
- L'**ereditarietà multipla**, invece, è una caratteristica in cui una classe può ereditare attributi e metodi **da più di una classe genitore**
- L'ereditarietà multipla comporta un **elevato livello di complessità** e **ambiguità** in situazioni come il **problema del diamante** (*diamond problem*)

Il problema del diamante o il "diamante mortale"

- Il "problema del diamante" (a volte indicato come "diamante mortale della morte") è il termine generalmente usato per indicare un'**ambiguità** che si presenta quando
  - Due **classi B** e **C** ereditano da una **superclasse A**
  - Un'altra classe D eredita sia da B che da C
  - Se c'è un **metodo "m"** in A per il quale B o C (o addirittura entrambi) hanno fatto un *overridden*, e se D non fa l'*overridden* di questo metodo, allora **quale versione del metodo eredita D?**

```
class A:
      def m(self):
      print("m of A called")

class B(A):
      def m(self):
      print("m of B called")

class C(A):
      def m(self):
      print("m of C called")

class D(B,C):
      pass
```

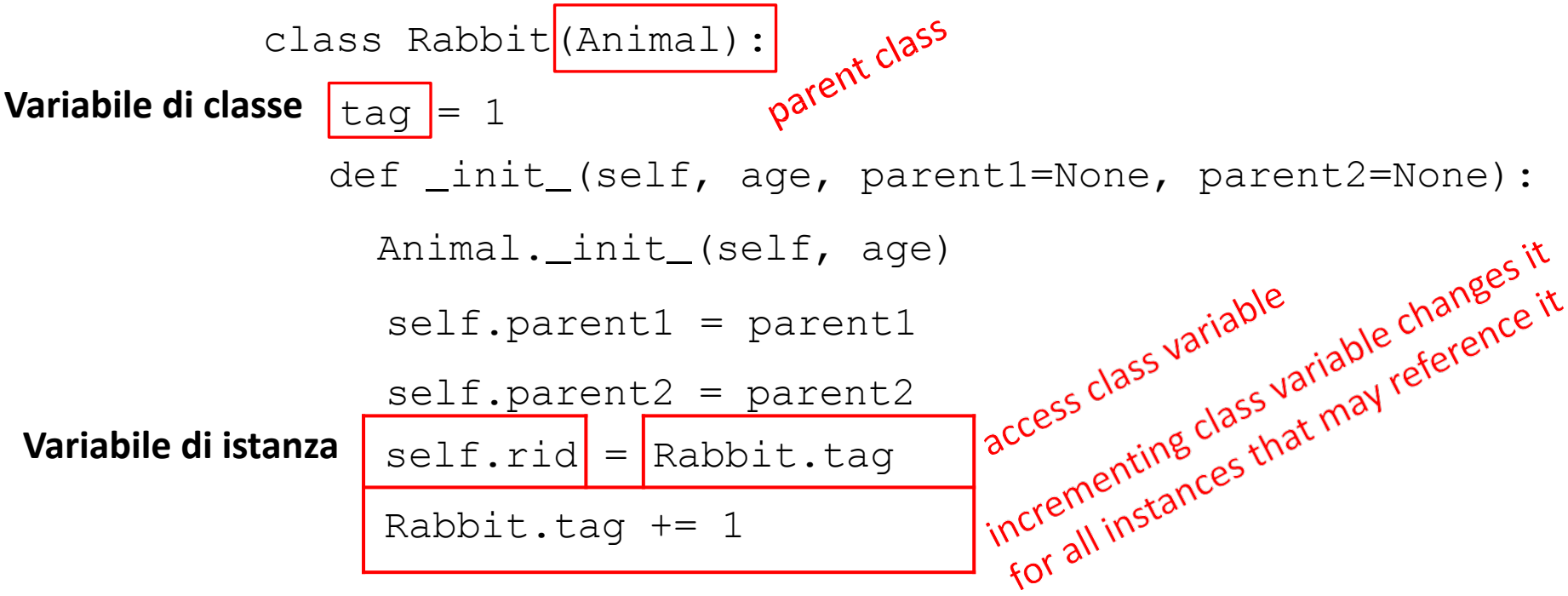
Il problema del diamante o il "diamante mortale"

```
a = A()
b = B()
c = C()
d = D()
print("Call m on object A: ")
a.m()
print("Call m on object B: ")
b.m()
print("Call m on object C: ")
c.m()
print("Call m on object D: ")
d.m() # ???
```

Le variabili statiche (di classe)

- Le **variabili statiche** in Python sono dette anche **variabili di classe,** e i loro valori sono condivisi tra tutte le istanze (oggetti) di una classe
- Possono essere **modificate dalla classe stessa** invece che attraverso un'istanza di tale classe
- **Caratteristiche**
  - Allocate in memoria una volta sola, quando l'oggetto della classe viene creato per la prima volta.
  - Create al di fuori dei metodi, ma all'interno di una classe.
  - Si può accedere alle variabili statiche attraverso una classe, ma non direttamente con un'istanza.
  - Il comportamento delle variabili statiche non cambia per ogni oggetto.

Le variabili statiche (di classe). Esempio



- Per esempio `tag` è utilizzato per dare un **id univoco** ad ogni nuovo oggetto istanza della classe

Variabili di classe. Esempio

```
class Rabbit(Animal):
    tag = 1
    def_init_(self, age, parent1=None, parent2=None):
        Animal._init_(self, age)
        self.parent1 = parent1
        self.parent2 = parent2
        self.rid = Rabbit.tag
        Rabbit.tag += 1
    def get_rid(self):
        return str(self.rid).zfill(3)
    def get_parent1(self):
        return self.parent1
    def get_parent2(self):
        return self.parent2
```

method on a string to pad the beginning with zeros for example, 001 not 1

- getter methods specific for a `Rabbit` class
- there are also getters `get_name` and `get_age` inherited from `Animal`

Decoratori in Python

- I **decoratori** sono uno strumento molto potente e utile in Python, poiché consentono ai programmatori di **modificare il comportamento di una funzione o di una classe**
- I decoratori permettono di fare il ***wrap*** di un'altra funzione per **estendere il comportamento della funzione** *wrapped*, senza modificarla in modo permanente!
- Nei decoratori, le funzioni vengono prese come argomento in un'altra funzione e poi richiamate all'interno della funzione *wrapper*

I metodi statici

- I **metodi statici** in Python sono metodi di una classe che possono essere invocati anche senza aver istanziato un oggetto di quella classe
- Tali metodi **non ricevono `self` come primo argomento**
- **Caratteristiche**
  - Definiti all'interno della classe, come gli altri metodi non statici, ma senza il parametro `self`
  - La definizione deve essere preceduta dal decoratore **`@staticmethod`**
  - Possono essere invocati direttamente sulla classe o su un oggetto della classe
  - Solitamente utilizzati per gestire aspetti non legati alle singole istanze di una classe.

Metodi statici. Esempio

```
class Rabbit(Animal):
    tag = 1
    def __init__(self, age, parent1=None, parent2=None):
        Animal.__init__(self, age)
        self.parent1 = parent1
        self.parent2 = parent2
        self.rid = Rabbit.tag
        Rabbit.tag += 1
    def get_rid(self):
        return str(self.rid).zfill(3)
    def get_parent1(self):
        return self.parent1
    def get_parent2(self):
        return self.parent2
```

```
@staticmethod
def get_tag():
    return Rabbit.tag
```

Static method with no `self` parameter

Return the current tag value, which is a static variable

Metodi statici. Esempio

```
r1 = Rabbit(3)
r2 = Rabbit(4)
r3 = Rabbit(5)

print("Call static get_tag method on class Rabbit:", Rabbit.get_tag())
print("Call static get_tag method on object r1: ", r1.get_tag())
```

Output:

```
Call static get_tag method on class Rabbit: 4
Call static get_tag method on object r1: 4
```

Overloading degli operatori in Python: **Esempio __add__**

```
def __add__(self, other):
        # returning object of same type as this class
        return Rabbit(0, self, other)
```

```
__init__(self, age, parent1=None, parent2=None)
```

All'atto del return invocherò la `__init__` di Rabbit

- Ridefinisco l'**operator +** tra due oggetti di tipo Rabbit
  - E' come se facessi **`r4 = r1 + r2`**
    - dove `r1` e `r2` sono istanze di `Rabbit`
    - `r4` è un'istanza di `Rabbit` con age = 0
    - `r4` ha l'oggetto `self` come primo parent e `other` come secondo parent
    - In `__init__`, `parent1` e `parent2` sono di tipo `Rabbit`

Overloading degli operatori in Python: **Esempio __eq__**

- Assumiamo che due `Rabbit` siano uguali se hanno gli stessi due genitori

booleans

```
def __eq__(self, other):
    parents_same = self.parent1.rid == other.parent1.rid \
                   and self.parent2.rid == other.parent2.rid
    parents_opposite = self.parent2.rid == other.parent1.rid \
                       and self.parent1.rid == other.parent2.rid
    return parents_same or parents_opposite
```

- Possiamo **comparare gli id dei genitori** dal momento che gli **ids sono univoci** (uso della variabile di classe tag)
- **Nota che non possiamo comparare gli oggetti direttamente!**
  - E.g., `self.parent1 == other.parent1` invoca a sua volta il metodo `__eq__` finchè non arriverò ad accedere all'attributo parent1 di un oggetto `None`, ritornando chiaramente un'eccezione `AttributeError` quando proverà a fare `None.parent1`

Eccezioni e ereditarietà

- Ogni eccezione è un oggetto della classe **`BaseException`** in Python o un oggetto di una classe che eredita da tale classe base
- Le classi di eccezione che ereditano direttamente o indirettamente dalla classe base **BaseException**, sono definite nel modulo **exceptions**

BaseException

- Python definisce quattro sottoclassi dalla classe base **BaseException**
  - **SystemExit** termina l'esecuzione del programma (o termina una sessione interattiva) e quando non rilevato non produce un traceback come altri tipi di eccezioni.
  - Si verificano eccezioni **KeyboardInterrupt** quando l'utente digita il comando Ctrl + C (o control + C) sulla maggior parte dei sistemi
  - Le eccezioni **GeneratorExit** si verificano quando un generatore si chiude, normalmente quando un generatore erator termina la produzione di valori o quando il suo metodo close viene chiamato in modo esplicito.
  - **Exception** è la classe base per le eccezioni più comuni. Sottoclassi di Exception sono ZeroDivisionError, NameError, ValueError, StatisticsError, TypeError, etc.

Definizione di Eccezioni: **Esempio**

```
class Error(Exception):
      """Base class for other exceptions"""
      pass

class ValueTooSmallError(Error):
      """Raised when the input value is too small"""
      pass

class ValueTooLargeError(Error):
      """Raised when the input value is too large"""
      pass
```

Polimorfismo in Python

Polimorfismo in Python

- Il polimorfismo consente alle **sottoclassi** di avere **metodi con gli stessi nomi dei metodi delle loro superclassi e invocare il metodo opportuno a *run-time***

- Python esegue il **controllo dei tipi a tempo di esecuzione**, a differenza dei linguaggi tipizzati staticamente (come Java) che lo eseguono a tempo di compilazione

Duck Typing in Python

“When I see a bird that walks like a duck and swims like a duck and quacks like a duck, I call that bird a duck.” [James Whitcomb Riley]

- Un caso particolare di *tipizzazione dinamica* in cui di utilizzano tecniche caratteristiche del polimorfismo, tra cui il **late binding** e il **dynamic dispatch**

- Nella tipizzazione normale, l'idoneità è determinata dal solo tipo di oggetto

- Con il *duck typing* si usa la presenza di metodi e proprietà per determinare l'idoneità piuttosto che il tipo effettivo dell'oggetto in questione.

Polimorfismo in Python: **Esempio**

```
animals = [a, c, p1, p2, s1, s2, r1, r2, r3, r4, r5, r6]
for animal in animals:
      print(animal)
...
animal::4
cat:fluffy:5
person:jack:30
person:jill:25
student:alice:20:CS
student:beth:18:None
rabbit:001
rabbit:002
rabbit:003
rabbit:004
rabbit:005
rabbit:006
```

Classi Astratte e Interfacce in Python

Classi Astratte e Interfacce in Python

- Una classe che contiene uno o più metodi astratti è chiamata **classe astratta**
- Un **metodo astratto** è un metodo che ha una dichiarazione ma **non ha un'implementazione**
- Una classe astratta può essere **considerata come un modello** per altre classi
  - Consente di creare un insieme di metodi che devono essere creati all'interno di ogni classe figlia costruita dalla classe astratta
- Le classi astratte **non possono essere istanziate**
  - Hanno bisogno di **sottoclassi che forniscano implementazioni per i metodi astratti** definiti nelle classi astratte
- Le classi astratte che hanno **solo metodi astratti** sono chiamate **interfacce**

Classi Astratte e il modulo **`abc`**

- Il modulo **`abc`** fornisce l'infrastruttura per la definizione di **Abstract Base Class (ABC)** in Python
- Questo modulo fornisce la metaclasse **`ABCMeta`** per definire le *Abstract Base Class* e una classe ausiliaria **`ABC`** per definire alternativamente le *Abstract Base Class* attraverso il meccanismo dell'ereditarietà

Classi Astratte e il modulo **abc**

```
from abc import ABC, abstractmethod

class Animal(ABC):
        @abstractmethod
        def doAction(self):
                pass

class Human(Animal):
        def doAction(self):
                print("I can walk and run")

class Snake(Animal):
        def doAction(self):
                print("I can crawl")

class Dog(Animal):
        def doAction(self):
                print("I can bark")

class Lion(Animal):
        def doAction(self):
                print("I can roar")
```

```
# A = Animal()
# a.doAction()
"""
questo scatena un
TypeError: Can't instantiate abstract class Animal with
abstract method doAction

perchè non posso istanziare un classe astratta
"""


R = Human()
R.doAction()

K = Snake()
K.doAction()

R = Dog()
R.doAction()

K = Lion()
K.doAction()
```

@abc.abstractmethod

- **@abc.abstractmethod** è il decoratore che permette di specificare che un **metodo è astratto**
- L'uso di questo decoratore richiede che la metaclasse della classe sia ABCMeta o che derivi da essa
- Una classe che ha una metaclasse derivata da ABCMeta non può essere istanziata a meno che tutti i suoi metodi astratti e le sue proprietà non vengano sovrascritti

NON GENERA ECCEZIONE

```
class Animal(ABC):
        def doAction(self):
                pass


A = Animal()
A.doAction()c
```

GENERA ECCEZIONE

```
class Animal(ABC):
        @abstractmethod
        def doAction(self):
                pass


A = Animal()
A.doAction()
```

Interfacce in Python

- Un'***interfaccia*** funge da modello per la progettazione delle classi e contiene **tutti metodi astratti** rispetto ad una classe astratta
  - Nota che una classe astratta può essere istanziata mentre un'interfaccia no!

- L'approccio di Python alla definizione di interfacce è un po' diverso rispetto a linguaggi come Java e C++
  - C++ e Java hanno una **parola chiave** per le interfacce, mentre **Python non ce l'ha**

- Python fornisce due approcci per la specifica di interfacce
  - *Interfacce informali*
  - *Interfacce formali* (uso di ABC)

Interfacce *informali*

- Il modo più semplice per definire un’interfaccia in Python è utilizzare il concetto di ***interfaccia informale***

- La natura dinamica di Python consente di implementare un'*interfaccia informale*, ovvero una classe che definisce dei metodi che possono essere ***overridden***, ma senza un'applicazione rigorosa

Interfacce *informali*. **Esempio**

```
class InformalInterface:
      def method1(self, arg1, arg2):
            pass
      def method2(self, arg1)
            pass
```

- **`InformalInterface`** definisce due metodi che non sono implementati, obbligando l'implementazione alle classi *concrete* che ereditano da **`InformalInterface`**

- **`InformalInterface`** è praticamente una normale classe Python, e si sfrutta il *duck typing* per informare gli utilizzatori della classe che si tratta di un'interfaccia e che deve essere usata di conseguenza.

Interfacce *informali*. **Esempio**

```
class InformalInterfaceImpl(InformalInterface):
      def method1(self, arg1, arg2):
            """ override method1 and implement it"""
      def method2(self, arg1)
            """ override method2 and implement it"""
```

- Per utilizzare l'interfaccia **`InformalInterface`**, è necessario creare una classe concreta, ovvero una sottoclasse dell'interfaccia che fornisce un'implementazione dei metodi dell'interfaccia.

Interfacce *formali*

- Per definire **interfacce formali** in Python si usa il modulo **`abc`** e si specificano **tutti metodi come astratti**
- Normalmente, si prevede sollevare un'**eccezione di non implementazione** (`NotImplementedError`) nel caso in cui la classe che implementa l'interfaccia non definisce un metodo astratto

Interfacce *formali*: **Esempio**

```
### interfaccia formale

import abc

class MyInterface(ABC):

        @abc.abstractmethod
        def metodo1(self):
                raise NotImplementedError

        @abc.abstractmethod
        def metodo2(self):
                raise NotImplementedError

        @abc.abstractmethod
        def metodo3(self):
                raise NotImplementedError
```

```
class MyInterfaceImpl(MyInterface):

        def metodo1(self):
                print("metodo1")

        def metodo2(self):
                print("metodo2")

        def metodo3(self):
                print("metodo3")

#c = MyInterface() # => TypeError: Can't
instantiate abstract class MyInterface with
abstract methods metodo1, metodo2, metodo3

c = MyInterfaceImpl() # => OK! se
implemento tutti i metodi astratti

c.metodo1()
c.metodo2()
c.metodo3()
```