University of Ferrara

Python

Ereditarietà

Ereditarietà

- Come gli altri linguaggi OOP, Python supporta l'ereditarietà tra classi.
- La superclasse dev'essere passata al momento della definizione della classe.

```
class MyClass(ItsSuperclass):
    # definizione
```

Ereditarietà

- Da notare che anche le classi che non specificano una classe nella loro definizione, ereditano di default da una classe.

```
class MyClass(object):
    # definizione
```

- Ogni classe è una sotto classe del tipo **`object`**
- Se noi omettiamo la superclasse, questa sarà **`object`**

Ereditarietà

- Come superclasse possiamo utilizzare espressioni arbitrarie. Questo potrebbe essere utile, ad esempio, quando la classe base è definite in un altro modulo

```
class MyClass(mymod.MySuperClass):
    # definizione
```

- Non ci sono meccanismi particolari per i costruttori.
- La ricerca di metodi/attributi viene fatta risalendo la catena delle superclassi.

Ereditarietà

```
class Pet:
    owner = "unknown"
    def __init__(self,id):
        print("pet")
        self.id = id

class Dog(Pet):
    def __init__(self, name):
        print("dog")
        self.name = name

p=Dog('doggo')
```

Non c'è nessuna chiamata al costruttore della superclasse. L'attributo **`id`** non sarà presente in **`p`**, mentre lo sarà **`owner`**.
Come possiamo istanziare in maniera corretta gli attributi di una superclasse?

Ereditarietà

```
class Pet:
    owner = "unknown"
    def __init__(self,id):
        print("pet")
        self.id = id

class Dog(Pet):
    def __init__(self, name, id):
        print("dog")
        Pet.__init__(self, id)
        self.name = name

p=Dog('doggo',2)
```

Usato in Python 2 e ancora valido in Python 3, ma…

Ereditarietà

```
class Pet:
    owner = "unknown"
    def __init__(self,id):
        print("pet")
        self.id = id

class Dog(Pet):
    def __init__(self, name, id):
        print("dog")
        super().__init__(id)
        self.name = name

p=Dog('doggo',2)
```

In Python 3 è meglio usare l'espressione super()

Ereditarietà

- Python ha due funzioni integrate che lavorano con l'ereditarietà:
- Si utilizza **`isinstance()`** per verificare il tipo di un'istanza:
- **`isinstance(obj, int)`** sarà **`True`** solo se **`obj.__class__`** è **`int`** oppure qualche classe derivate da **`int`**.
  - Si usa **`issubclass()`** per verificare la classe padre: **`issubclass(bool, int)`** è **`True`** perchè **`bool`** è una sottoclasse di **`int`**. Di contro, **`issubclass(float, int)`** è **`False`** in quanto **`float`** non è sottoclasse di **`int`**

Ereditarietà Multiclasse

- Python supporta l'ereditarietà multipla. La definizione di classe diventa:

```
class MyClass(Class1,Class2):
    # definizione
```

- La ricerca di attributi ereditati da una classe genitore procede con una ricerca in profondità, da sinistra a destra, senza ripetere due volte la ricerca all'interno della stessa classe, quando ci fosse una sovrapposizione nella gerarchia.
  - Se un attributo non è in **`MyClass`** viene cercato in **`Class1`** (se non fosse presente, in una sua super classe). Se non fosse presente in **`Class1`** e nelle sue superclassi, verrà cercato in **`Class2`** e nelle sue superclassi.

Ereditarietà Multiclasse

- La ricerca di metodi ed attributi è eseguita secondo un ordine prestabilito e impedendo la ricerca nella stessa classe per più di una volta. Questo metodo è detto **Method Resolution Order (MRO)**.
- E' possibile vedere l'elenco delle classi ricercate usando l'attributo `__mro__`
- La prima corrispondenza trovata interrompe la ricerca.

Ereditarietà Multiclasse

```
class A:
    pass

class B(A):
    pass

class C(A):
    pass

class E(B,C):
    pass

print(E.__mro__)
```

Output:

```
(<class '__main__.E'>,
 <class '__main__.B'>,
 <class '__main__.C'>,
 <class '__main__.A'>,
  <class 'object'>)
```

Ereditarietà Multiclasse

```
class A:
    pass

class B(A):
    pass

class C(A):
    pass

class E(B,C):
    pass

print(E.__mro__)
```

Output:

```
(<class '__main__.E'>,
 <class '__main__.B'>,
 <class '__main__.C'>,
 <class '__main__.A'>,
  <class 'object'>)
```

Come si può notare, Python non sempre segue una strategia di ricerca in profondità, da sinistra a destra. Prima crea la catena completa **E – B – A – Object – C – A – Object** poi rimuove le classi (da sinistra a destra) tali che ci sia una classe nella coda di ricerca che eredita da essa. In questo caso è più naturale usare il metodo definito dalla sua classe derivata.

Ereditarietà Multiclasse

- Cosa succede quando più superclassi hanno metodi con lo stesso nome?

```
class A:
    def method(self):
        pass

class B(A):
    def method(self):
        pass

class C(A,B):
    def method1(self):
        super().method()
```

Seguendo l' MRO, il metodo **`method`** di **`A`** viene invocato. Cosa dovremmo fare per invocare il metodo **`method`** di **`B`**?

Ereditarietà Multiclasse

- Cosa succede quando più superclassi hanno metodi con lo stesso nome?

```
class A:
    def method(self):
        pass

class B(A):
    def method(self):
        pass

class C(A,B):
    def method1(self):
        B.method(self)
```

Soluzione: esplicitiamo la classe di cui vogliamo invocare il metodo.

MRO

- Basato su C3, ordine per risolvere metodi. https://en.wikipedia.org/wiki/C3_linearization.
- C3 è stato inventato da persone che stavano lavorando su Dylan
- Linearizzazione degli antenati che soddisfa
  - **Monotonicità**: un MRO è monotono quando è vero quanto segue: *se C1 precede C2 nella linearizzazione di C, allora C1 precede C2 nella linearizzazione di una qualunque sottoclasse di C.*
  - **Ordinamento di precedenza locale**: l'ordine nella lista delle precedenze locali, come la lista dei genitori di G, dev'essere preservato nella linearizzazione di G.

Variabili private

- Python non dà la possibilità di creare variabili e metodi accessibili solo dalla classe nella quale sono definiti.

  Ricorda, Python si basa su convenzioni!

- Un nome preceduto dal trattino basso dev'essere considerate privato → `_private_id`
- Questo è alla base del meccanismo di nomenclatura usato dagli esperti di programmazione, chiamato *name mangling*.
- Variabili e metodi "privati" rimangono accessibili, ma per convenzione non li si utilizza.
- Vedi https://www.geeksforgeeks.org/private-variables-python/ per una spiegazione di questo concetto!

Name Mangling

- Qualsiasi identificatore della forma__geek (almeno due caratteri di sottolineatura iniziali e al massimo un carattere di sottolineatura finale) viene sostituito con _classname__geek, dove classname è il nome della classe corrente senza trattini bassi iniziali.
- Il name mangling avviene solo all'interno della definizione della classe.
- In pratica si usa per gli attributi e i metodi che si vuole che non vengano usati nelle sottoclassi

Name Mangling

```
class Test:
    def __init__(self):
        self.foo = 11
        self._bar = 23
        self.__baz = 23
```

`dir([object])`
Senza argomenti ritorna la lista dei nomi presenti all'interno dello scope locale.
Con un argomento, tenta di restituire un elenco di attributi validi per quell'oggetto.

```
>>> t = Test()
>>> dir(t)
['_Test__baz', '__class__', '__delattr__', '__dict__', '__dir__',
 '__doc__', '__eq__', '__format__', '__ge__', '__getattribute__',
 '__gt__', '__hash__', '__init__', '__le__', '__lt__', '__module__',
 '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__',
 '__setattr__', '__sizeof__', '__str__', '__subclasshook__',
 '__weakref__', '_bar', 'foo']
```

Name Mangling

```
class ExtendedTest(Test):
    def __init__(self):
        super().__init__()
        self.foo = 'overridden'
        self._bar = 'overridden'
        self.__baz = 'overridden’
>>> t2 = ExtendedTest()
>>> t2.foo
'overridden'
>>> t2._bar
'overridden'
>>> t2.__baz
AttributeError: "'ExtendedTest' object has no attribute '__baz’"
>>> t2._ExtendedTest__baz
'overridden'
```

Name Mangling

```
>>> t2._ExtendedTest__baz
'overridden'
>>> t2._Test__baz
23
```

Convenzione dei Nomi

- **Singolo trattino basso iniziale: _var.** Il trattino basso posto come prefisso è inteso come suggerimento per un altro programmatore che la variabile o il metodo sono destinati ad un uso interno.
- **Singolo trattino basso finale: var_.** A volte il nome più appropriato per una variabile è già preso da una parola chiave. Pertanto nomi come class o def non possono essere utilizzati come nomi di variabili in Python. In questo caso puoi aggiungere un trattino basso per evitare il conflitto di denominazione

Convenzione dei Nomi

- **Double Leading and Trailing Underscore: __var__.** Il name mangling *non* viene applicato se un nome *inizia* e *finisce* con il doppio trattino basso
- I nomi che hanno il doppio trattino basso sia all'inizio che alla fine sono riservati ad usi speciali del linguaggio (es. __init__)

Convenzione dei Nomi

- **Singolo trattino basso: _.** Un singolo trattino basso da solo si usa per indicare che una variabile è temporanea o insignificante

```
>>> for _ in range(32):
...     print('Hello, World.')
```

Convenzione dei Nomi

- Si può usare il singolo trattino basso con l'obiettivo di indicare che non si è interessati a quella particolare varibile ed al suo valore

```
>>> car = ('red', 'auto', 12, 3812.4)
>>> color, _, _, mileage = car

>>> color
'red'
>>> mileage
3812.4
>>> _
12
```

Metodi statici

- Un metodo statico non riceve implicitamente il primo argomento come I metodi standard.
- Per creare un metodo statico si deve utilizzare il simbolo **`@staticmethod`** (è un decorator)
- In Python i metodi statici possono essere chiamati indifferentemente sule classi o sulle istanze.
- Grazie al decorator, se chiamato su un'istanza, non viene aggiunto l'argomento **`self`**

```
class Point:
    @staticmethod
    def distance(p1, p2):
        # istruzioni
```

Esercizio 6

- Creare la classe **Vehicle** con 2 attributi speed ed acceleration (float)
- Il **costruttore** istanzia i 2 attributi, ma potrebbe anche non ricevere argomenti in ingresso (in caso di assenza di argomenti li inizializzerà a 0.0)
- Implementare 2 metodi **set_speed** e **set_acceleration** che prendono in ingresso un valore (che può anche essere una stringa) e settano il corrispondente attributo al valore passato.
- Si implementi il metodo **print_speed** e **print_acceleration** che stampi a video i 2 valori.
- Si implementi il **metodo statico compute_speed_increment** che prenda in ingresso un float per l'accelerazione ed un int che rappresenti i secondi e che restituisca l'incremento della velocità ottenuta dall'accelerazione per i secondi dati: accelerazione*secondi.

Esercizio 6

- Creare una classe **Car sottoclasse di Vehicle** con due **attributi di istanza** plate (stringa) e running (booleano)
- Il **costruttore** istanzia i 4 attributi. Deve prendere in ingresso plate (nessun valore di default) mentre gli altri valori possono avere valori di default (il default per running è False). Prova a passare al costruttore super solo i valori che vengono dati come input al costruttore di Car. Se running è False, settare velocità ed accelerazione a 0.
- Implementare 2 metodi **start** e **stop** che modifichino il valore di running. Il metodo stop deve anche resettare velocità ed accelerazione.
- Implementare il metodo **accelerate** che prende l'accelerazione (float) e i secondi (int) ricevuti in input ed utilizza il metodo statico della classe Veicolo per calcolare l'incremento della velocità e sommare questo valore alla velocità dell'istanza. Alla fine, assegna il valore ottenuto. Questo metodo funziona solo se running è True.
- Implementa il **metodo static print_n_wheels** che stampi a video il numero di route di una macchina (4).

Esercizio 6

- Creare la classe **Bicycle sottoclasse di Vehicle**
- Il **costruttore** istanzia gli attributi velocità e accelerazione. I due attributi possono essere passati o meno. Provare a passare al costruttore super solo i valori che sono stati dati in ingresso al costruttore di biciclette.
- Implementa il metodo **pedal** che prende il numero di pedalate **n_hits** (int) ed i **secondi** (int) e che calcola l'accelerazione come n_hits/sec^2, imposta la nuova accelerazione e la nuova velocità aggiungendo ai valori degli attributi di istanza l'incremento ottenuto usando il metodo statico di Car.
- Implementare un **metodo statico print_n_wheels** che stampi a video il numero di ruote di Bicycle (2).

Esercizio 6

- Istanzia un oggetto di tipo Car ed uno di tipo Bicycle ed utilizza i loro metodi per verificare se sono ben implementati usando il codice

```
c = Car('ABC',False,10,11)
b = Bicycle('10.1',’0')
Car.print_n_wheels()
b.print_n_wheels()
b.set_speed(5)
c.set_acceleration(1)
print(" - Car c ----------")
c.print_speed()
c.print_acceleration()
print("\n\n - Bicycle b ------")
b.print_speed()
b.print_acceleration()
print(" ------------------")
c.start()
c.accelerate(10.1,2)
b.pedal(2,5)
print(" - Car c ----------")
c.print_speed()
c.print_acceleration()
print("\n\n - Bicycle b ------")
b.print_speed()
b.print_acceleration()
print(" ------------------")
c.stop()
print(" - Car c ----------")
c.print_speed()
c.print_acceleration()
print("\n\n - Bicycle b ------\n")
b.print_speed()
b.print_acceleration()
print(" ------------------\n")
```

Esercizio 6

- Deve stampare

```
I have 4 wheels!
I have 2 wheels!
 - Car c ----------
Speed:  0.0
Acceleration: 0.0
 - Bicycle b ------
Speed:  5
Acceleration: 0.0
 ------------------
 - Car c ----------
Speed:  20.2
Acceleration: 10.1
 - Bicycle b ------
Speed:  5.4
Acceleration: 0.08
 ------------------
 - Car c ----------
Speed:  0.0
Acceleration: 0.0
 - Bicycle b ------
Speed:  5.4
Acceleration: 0.08
 ------------------
```

Basato su slide dell'Ing. Riccardo Zese
riccardo.zese@unife.it