Programmazione avanzata a.a. 2024-25

A. De Bonis

82

I metodi statici e i metodi di classe

- Un metodo di una classe normalmente riceve un'istanza della classe come primo argomento
- A volte però i programmi necessitano di elaborare dati associati alle classi e non alle loro istanze.
  - Ad esempio tenere traccia del numero di istanze della classe create
- Per questi scopi potrebbe essere sufficiente scrivere funzioni esterne alla classe perché queste funzioni possono accedere agli attributi della classe attraverso il nome della classe stessa.
- Per associare meglio la funzione alla classe e per fare in modo che la funzione venga ereditata dalle sottoclassi ed eventualmente ridefinita in esse, è meglio codificare le funzioni all'interno delle classi
- Abbiamo però bisogno di metodi che non si aspettano di ricevere self come argomento e quindi funzionano indipendentemente dal fatto che esistano istanze della classe

83

I metodi statici e i metodi di classe

Python permette di definire

- **Metodi statici.** I metodi statici non ricevono self come argomento sia nel caso in cui vengano **invocati su una classe**, sia nel caso in cui vengano **invocati su un'istanza della classe**. Di solito tengono traccia di informazioni che riguardano tutte le istanze piuttosto che fornire funzionalità per le singole istanze
- **Metodi di classe**. I metodi di classe ricevono un oggetto classe come primo argomento invece che un'istanza, sia che vengano invocati su una classe, sia nel caso in cui vengano invocati su un'istanza della classe. Questi metodi possono accedere ai dati della classe attraverso il loro argomento cls (corrisponde all'argomento self dei metodi "di istanza")

84

I metodi statici e i metodi di classe

- la funzione printNumIstances (non è né un metodo di classe né un metodo statico) non utilizza informazioni delle istanze ma solo informazioni della classe
- Vogliamo quindi invocarla senza far riferimento ad una particolare istanza
  - creare un'istanza solo per invocare la funzione farebbe aumentare il numero di istanze

spam.py

```
class Spam:
    numInstances = 0
    def __init__(self):
        Spam.numInstances = Spam.numInstances + 1
    def printNumInstances():
        print("Number of instances created: %s" % Spam.numInstances)
```


85

I metodi statici e i metodi di classe

- In Python 3.X è possibile invocare funzioni senza l'argomento self se le invochiamo attraverso la classe **e non attraverso un'istanza**

```
>>> from spam import Spam
>>> a = Spam()                          # Can call functions in class in 3.X
>>> b = Spam()                          # Calls through instances still pass a self
>>> c = Spam()

>>> Spam.printNumInstances()            # Differs in 3.X
Number of instances created: 3
>>> a.printNumInstances()
TypeError: printNumInstances() takes 0 positional arguments but 1 was given
```


86

I metodi statici e i metodi di classe

- I metodi statici si definiscono invocando la funzione built-in **staticmethod**
- I metodi di classe si definiscono invocando la funzione built-in **classmethod**

87

I metodi statici e i metodi di classe

```
# File bothmethods.py

class Methods:
    def imeth(self, x):              # Normal instance method: passed a self
        print([self, x])

    def smeth(x):                    # Static: no instance passed
        print([x])

    def cmeth(cls, x):               # Class: gets class, not instance
        print([cls, x])

    smeth = staticmethod(smeth)      # Make smeth a static method (or @: ahead)
    cmeth = classmethod(cmeth)       # Make cmeth a class method (or @: ahead)
```

```
>>> Methods.smeth(3)
[3]
>>> obj.smeth(4)
[4]
```

```
>>> Methods.cmeth(5)
[<class 'bothmethods.Methods'>, 5]
>>> obj.cmeth(6)
[<class 'bothmethods.Methods'>, 6]
```


88

Metodo statico che conta le istanze

spam_static.py

```
class Spam:
    numInstances = 0                          # Use static method for class data
    def __init__(self):
        Spam.numInstances += 1
    def printNumInstances():
        print("Number of instances: %s" % Spam.numInstances)
    printNumInstances = staticmethod(printNumInstances)
```

```
>>> from spam_static import Spam
>>> a = Spam()
>>> b = Spam()
>>> c = Spam()
>>> Spam.printNumInstances()
Number of instances: 3
>>> a.printNumInstances()
Number of instances: 3
```


89

Metodo statico che conta le istanze

spam_static.py

```
class Sub(Spam):
    def printNumInstances():                    # Override a static method
        print("Extra stuff...")                 # But call back to original
        Spam.printNumInstances()
    printNumInstances = staticmethod(printNumInstances)
```

```
>>> from spam_static import Spam, Sub
>>> a = Sub()
>>> b = Sub()
>>> a.printNumInstances()                       # Call from subclass instance
Extra stuff...
Number of instances: 2
>>> Sub.printNumInstances()                     # Call from subclass itself
Extra stuff...
Number of instances: 2
>>> Spam.printNumInstances()                    # Call original version
Number of instances: 2
```


Metodo di classe che conta le istanze

spam_class.py

```
class Spam:
    numInstances = 0                            # Use class method instead of static
    def __init__(self):
        Spam.numInstances += 1
    def printNumInstances(cls):
        print("Number of instances: %s" % cls.numInstances)
    printNumInstances = classmethod(printNumInstances)
```

```
>>> from spam_class import Spam
>>> a, b = Spam(), Spam()
>>> a.printNumInstances()
Number of instances: 2
>>> Spam.printNumInstances()
Number of instances: 2
```

Metodo di classe che conta le istanze

Attenzione: Quando si usano i metodi di classe essi ricevono la classe più in basso dell'oggetto attraverso il quale viene invocato il metodo

```
class Spam:
    numInstances = 0                        # Trace class passed in
    def __init__(self):
        Spam.numInstances += 1
    def printNumInstances(cls):
        print("Number of instances: %s %s" % (cls.numInstances, cls))
    printNumInstances = classmethod(printNumInstances)

class Sub(Spam):
    def printNumInstances(cls):             # Override a class method
        print("Extra stuff...", cls)        # But call back to original
        Spam.printNumInstances()
    printNumInstances = classmethod(printNumInstances)
```


92

Metodo di classe che conta le istanze

spam_class.py

```
class Spam:
    numInstances = 0                        # Trace class passed in
    def __init__(self):
        Spam.numInstances += 1
    def printNumInstances(cls):
        print("Number of instances: %s %s" % (cls.numInstances, cls))
    printNumInstances = classmethod(printNumInstances)

class Sub(Spam):
    def printNumInstances(cls):             # Override a class method
        print("Extra stuff...", cls)        # But call back to original
        Spam.printNumInstances()
    printNumInstances = classmethod(printNumInstances)

class Other(Spam): pass                     # Inherit class method verbatim
```


93

Metodo di classe che conta le istanze

```
>>> from spam_class import Spam, Sub, Other
>>> x = Sub()
>>> y = Spam()
>>> x.printNumInstances()                    # Call from subclass instance
Extra stuff... <class 'spam_class.Sub'>
Number of instances: 2 <class 'spam_class.Spam'>
>>> Sub.printNumInstances()                  # Call from subclass itself
Extra stuff... <class 'spam_class.Sub'>
Number of instances: 2 <class 'spam_class.Spam'>
>>> y.printNumInstances()                    # Call from superclass instance
Number of instances: 2 <class 'spam_class.Spam'>
>>> z = Other()                              # Call from lower sub's instance
>>> z.printNumInstances()
Number of instances: 3 <class 'spam_class.Other'>
```


Metodo di classe invocato attraverso le sottoclassi

le sottoclassi hanno la propria variabile numInstances

spam_class2.py

```
class Spam:
    numInstances = 0
    def count(cls):                       # Per-class instance counters
        cls.numInstances += 1             # cls is lowest class above instance
    def __init__(self):
        self.count()                      # Passes self.__class__ to count
    count = classmethod(count)

class Sub(Spam):
    numInstances = 0
    def __init__(self):                   # Redefines __init__
        Spam.__init__(self)

class Other(Spam):                        # Inherits __init__
    numInstances = 0
```

Metodo di classe invocato attraverso le sottoclassi

```
class Spam:
    numInstances = 0
    def count(cls):                     # Per-class instance counters
        cls.numInstances += 1           # cls is lowest class above instance
    def __init__(self):
        self.count()                    # Passes self.__class__ to count
    count = classmethod(count)

class Sub(Spam):
    numInstances = 0
    def __init__(self):                 # Redefines __init__
        Spam.__init__(self)

class Other(Spam):                      # Inherits __init__
    numInstances = 0
```

```
>>> from spam_class2 import Spam, Sub, Other
>>> x = Spam()
>>> y1, y2 = Sub(), Sub()
>>> z1, z2, z3 = Other(), Other(), Other()
>>> x.numInstances, y1.numInstances, z1.numInstances          # Per-class data!
(1, 2, 3)
>>> Spam.numInstances, Sub.numInstances, Other.numInstances
(1, 2, 3)
```


Alternativa per definire metodi statici e metodi di classe

- I metodi statici e i metodi di classe possono essere definiti usando i seguenti decoratori
  - @staticmethod
  - @classmethod

```
@staticmethod
def smeth(x):
    print([x])

@classmethod
def cmeth(cls, x):
    print([cls, x])
```

Cenni sui decoratori di funzioni

- Specificano comportamenti speciali per le funzioni e i metodi delle classi.
- Creano intorno alla funzione un livello extra di logica implementato da un'altra funzione chiamata metafunzione (funzione che gestisce un'altra funzione)
- Da un punto di vista sintattico, un decoratore di funzione è una sorta di dichiarazione riguardante la funzione che viene avviene durante l'esecuzione del programma. Un decoratore è specificato su una linea che precede lo statement def e consiste del simbolo @ seguito da una metafunzione
- Il decoratore di funzione può restituire la funzione originale così come è oppure restituire un nuovo oggetto che fa in modo che la funzione originale venga invocata indirettamente dopo aver eseguito il codice della metafunzione

98

Cenni sui decoratori di funzioni

```
class C:
   @staticmethod                    # Function decoration syntax
   def meth():
       ...
```

è equivalente a

```
class C:
   def meth():
       ...
   meth = staticmethod(meth)        # Name rebinding equivalent
```


99

__slots__

- In Python ogni istanza di una classe ha un dizionario (__dict__) che memorizza gli attributi
- Considerando MyClass ed una sua istanza var_c, provate ad eseguire
  - print(MyClass.__dict__)
  - print(var_c.__dict__)
- Spreco di spazio se la classe ha pochi attributi
  - Problema aggravato se si creano tante istanze della classe
- Si può sovrascrivere il comportamento di default definendo __slots__ quando si definisce una classe

100

__slots__

- A __slots__ si assegna una sequenza di variabili di istanza ed è riservato, in ogni istanza della classe, solo lo spazio sufficiente a memorizzare un valore per ogni variabile
  - __dict__ non sarà più creato
  - non sono più possibili assegnamenti dinamici
    - superabile con __slots__ = . . . , '__dict__'

101

__slots__

```
>>> class MyNewClass:
...     __slots__='L'
...     def __init__(self,*args):
...         self.L=args
...
>>> var_cn=MyNewClass(1,2,3)
>>> var_cn.L
(1, 2, 3)
>>> var_cn.L=[4,5]
>>> var_cn.L
[4, 5]
```

continua nel riquadro a destra

```
>>> var_cn.X=3
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'MyNewClass' object has no attribute 'X'
>>> var_cn.__slots__
'L'
>>> var_cn.__dict__
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'MyNewClass' object has no attribute '__dict__'
>>> MyNewClass.__slots__
'L'
```


Riferimenti

- The Python Tutorial https://docs.python.org/3/tutorial/
- M.T. Goodrich, R. Tamassia, M.H. Goldwasser Data Structures and Algorithms in Python Capitolo 2, Object-Oriented Programming

- Studiare anche le sezioni
  - 2.3.3 Multidimensional Vector Class
  - 2.3.5 Range Class
  - 2.4.2 Hierarchy of Numeric Progressions