Programmazione avanzata a.a. 2020-21

A. De Bonis

Introduzione a Python (VII e VIII parte)

50

Ereditarietà

- Supportata da Python come segue

```
class DerivedClassName(BaseClassName):
    <statement-1>
    . . .
    <statement-N>
```

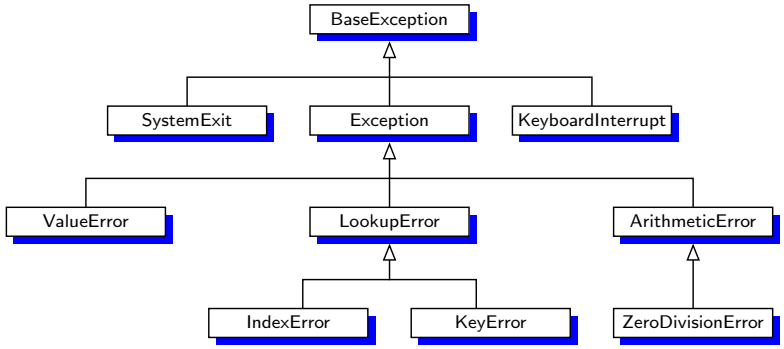
- BaseClassName deve essere definita nello scope che contiene la definizione della classe derivata DerivedClassName
- Si possono usare classi base definite in altri moduli

```
class DerivedClassName(modname.BaseClassName):
```


51

Gerarchia delle eccezioni in Python

Piccola porzione della gerarchia



52

Ereditarietà

- Le classi derivate possono
  - aggiungere variabili di istanza
  - sovrascrivere i metodi della classe base
  - accedere ai metodi e variabili della classe base
- Python supporta l'ereditarietà multipla

```
class DerivedClassName(Base1, Base2, Base3):
    <statement-1>
    .
    .
    <statement-N>
```


53

Esempio

```
class Base:
   def __init__(self, a, b):
      self._a = a
      self._b = b

   def stampa(self):
      print('<{0}>,<{1}>'.format(self._a, self._b))


class Derivata(Base):
   def __init__(self, a, b, c):
      super().__init__(a, b)
      self._c = c

   def stampa(self):
      print('[{0}]-[{1}]'.format(self._a, self._b))
```

Base.__init__(self,a,b)

```
b = Base(1, 3)
b.stampa()
d = Derivata('a', 'b', 9)
d.stampa()
```

<1>,<3>
[a]-[b]

Utilizzo metodi classe base

- Per invocare metodi di istanza definiti nella classe base si usa la funzione super()
  - super().nomemetodo(argomenti)
- Oppure si usa

  BaseClassName.nome_metodo(self, argomenti)
  - Funziona quando BaseClassName è accessibile come BaseClassName nello scope globale

super()

- Serve per accedere ai metodi della classe base che sono stati sovrascritti con la derivazione
  - super() restituisce un riferimento ad un oggetto
- Un altro modo per far riferimento, tramite super, a metodi sovrascritti è
  - super(DerivedClassName, self).nome_metodo(parametri)

56

super()

```
class base:
    def f(self):
        print("base")

class der(base):
    def f(self):
        print("der")

    def g(self):
        self.f()
        super().f()
        super(der,self).f()
        base.f(self)

class derder(der):
    def f(self):
        print("derder")
    def h(self):
        self.f()
        super().f()
        super(derder,self).f()
        super(der,self).f()
```

```
x=der()
x.g()
y=derder()
y.h()
```

```
der
base
base
base
derder
der
der
base
```


57

super()

```
class base:
   def __init__(self,v):
      self.a=v
   def f(self):
         print("base --","a =",self.a)

class der(base):
   def f(self):
         print("der -- ","a =",self.a)

class derder(der):
   def f(self):
      print("derder -- ", "a =",self.a)


x=der(10)
super(der,x).f()
print()
y=derder(20)
super(derder,y).f()
super(der,y).f()
```

```
base -- a = 10

der --  a = 20
base -- a = 20
```


58

Ereditarietà Multipla

- Nella classe derivata la ricerca degli attributi ereditati da una classe genitore avviene
  - dal basso verso l'alto e da sinistra verso destra

class DerivedClassName(Base1, Base2, Base3):

- Se un attributo non è trovato in DerivedClassName lo si cerca in Base1, dopo (ricorsivamente) nelle classi base di Base1 e, se non viene trovato si procede con Base2 e così via

59

Ereditarietà Multipla

- L'attributo__mro__ di una classe contiene l'elenco delle classi in cui si cerca il metodo che è stato invocato su un'istanza della classe
  - Le classi sono esaminate secondo l'ordine indicato in __mro__ (mro: Method Resolution Order)
    - L'attributo dipende da come è stata definita la classe
    - Il metodo mro() è invocato quando si crea un'istanza della classe . Questo metodo può essere sovrascritto (in una metaclasse) per modificare l'ordine in cui vengono cercati i metodi nelle classi che formano la gerarchia. L'ordine stabilito da mro() è memorizzato in __mro___ .
    - L'attributo è a sola lettura

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

D.__mro__

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

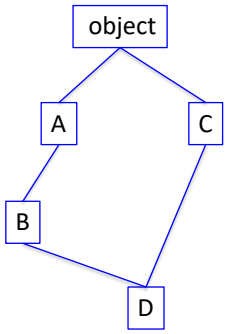

60

```
class A():
  pass

class B(A):
   pass

class C():
   pass

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



D.__mro__

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


61

Attributo __bases__

- Contiene la tupla delle classi base di una classe
  - Accessibile in lettura/scrittura
  - Modificando __bases__ l'attributo __mro__ è *ricomputato*
- Per modificare __bases__ si usa la funzione setattr
  - setattr(Derivata, '__bases__', (Base2, Base1))

62

Funzioni built-in

- **isinstance**(ist, classe) serve per verificare il tipo di un'istanza di una classe

```
f = 3.4
print(isinstance(f, float))
```
⟶ **True**

- **issubclass**(x,y) serve per verificare se x è una sottoclasse di y

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

issubclass(A,C) ⟶ **False**

issubclass(D,C) ⟶ **True**

63

Ordine differente rispetto a mro

- Assumiamo che le classi A, B e C definiscano il metodo metodo_base e che D sia derivata da A, B e C (**class** D(A,B,C): pass) e sia d un'istanza di D
- Se si esegue d.metodo_base(parametri), allora è eseguito metodo_base definito nella classe A
- Per invocare metodo_base definito in un'altra classe base si deve far riferimento direttamente alla classe base specifica

64

```
class A():
    def __init__(self, a, val):
        self._a = a
        self._val = val

    def stampa(self):
        print('a =', self._a, 'val =',self._val)

class B():
    def __init__(self, b, val):
        self._b = b
        self._val = val

    def stampa(self):
        print('b =', self._b, 'val =',self._val)

class C():
    def __init__(self, c, val):
        self._c = c
        self._val = val

    def stampa(self):
        print('c =', self._c, 'val =',self._val)
```

```
class D(A,B,C):
    def __init__(self, a, b, c, val):
        A.__init__(self, a, val)
        B.__init__(self, b, 2*val)
        C.__init__(self, c, 3*val)

    def stampa(self):
        C.stampa(self)
        B.stampa(self)
        A.stampa(self)
```

```
d = D(1,2,3, 123)
d.stampa()
```

```
c = 3 val = 369
b = 2 val = 369
a = 1 val = 369
```


65

Iteratori

- Se una classe supporta l'iteratore possiamo ottenere un riferimento ad esso tramite la funzione iter()
  - Si invoca iter su un'istanza della classe
- Per ottenere il prossimo elemento nella classe invochiamo next() sull'iteratore ottenuto
- Viene lanciata un'eccezione quando non ci sono più elementi nell'istanza della classe

66

```
lista = [85,23,59]
print(lista)
it = iter(lista)
print(it)
print(next(it))
print(next(it))
print(next(it))
```

→

```
[89, 23, 59]
<list_iterator object at 0x10217aa58>
89
23
59
```

```
lista = [59, 42, 90]
print(lista)
it = iter(lista)
print(it)
print(next(it))
print(next(it))
print(next(it))
print(next(it))
```

→

```
[59, 42, 90]
<list_iterator object at 0x10ad62908>
59
42
90
Traceback (most recent call last):
  File "/Users/adb/Documents/r.py", line 8, in
<module>
    print(next(it))
StopIteration
```

← **eccezione**

67

Gestire l'eccezione

```
lista=[59, 42, 90]
print(lista)
it = iter(lista)
print(it)

while True:
    try:
        print(next(it))
    except Exception as e:
      break
```


68

Generatori

- Una funzione generatore è un modo semplice ed immediato per creare un iteratore (detto generatore)
  - I metodi del generatore **__iter__()** e **__next__()** sono creati automaticamente
  - **__iter__() :** restituisce l'iteratore stesso
  - **__next__() :** viene utilizzato nei cicli per ottenere il prossimo elemento
- La sintassi per definire una funzione generatore è simile a quella usata per definire una funzione, ma al posto di **return** si usa **yield**
- Quando si incontra un **yield** l'esecuzione del generatore è sospesa, viene restituito il valore indicato da **yield**
- Ogni volta che si chiama **next()**, il generatore riparte da dove l'esecuzione era stata sospesa
  - Si parte dall'istruzione successiva a **yield**

69

Generatori

```
def gen_range(n):
    k=0
    while k<n:
        yield k
        k += 1
```

```
for i in gen_range(10):
    print(i, end=' ')
```

0 1 2 3 4 5 6 7 8 9

```
ite=gen_range(10)
while(True):
    try:
        i=next(ite)
        print(i, end=' ')
    except Exception as e:
        break
```

0 1 2 3 4 5 6 7 8 9

Si tratta di una sorta di funzione che genera una sequenza di valori restituiti uno per volta tramite **yield**

Nel generatore non possono coesistere **yield** e **return**

70

Generatori

```
def reverse(data):
    for index in range(len(data)-1, -1, -1):
        yield data[index]

for char in reverse('programmazione'):
    print(char, end='')
```

enoizammargorp

71

Superclassi astratte

- Una superclasse astratta è una classe il cui comportamento è in parte specificato dalle sottoclassi

- Se un metodo che deve essere definito dalle sottoclassi non è definito in una sottoclasse allora Python lancia un'eccezione quando effettua la ricerca del metodo nella gerarchia delle classi

72

Superclassi astratte

- A volte i programmatori, per rendere più evidente ciò che deve essere specificato nelle sottoclassi, usano degli **statement assert** o degli statement raise che lanciano l'eccezione NotImplementedError

Uso di assert nella funzione che deve essere fornita dalle sottoclassi

```
>>> class Super:
...     def delegate(self):
...             self.action()
...     def action(self):
...             assert False, 'action must be defined!'
…
>>> X = Super()
>>> X.delegate()
AssertionError: action must be defined!
```


73

Assert statement

- Rappresentano un modo per inserire asserzioni per il debugging in un programma
  - **assert** expression

  è equivalente a
  - **if** __debug__:
    if not expression: **raise** AssertionError

  - assert expression1, expression2

  è equivalente a
  - **if** __debug__:
    if not expression1: **raise** AssertionError(expression2)
- Negli if in alto AssertionError è l'eccezione built-in lanciata quando uno statement assert fallisce e __debug__ è una variabile built-in
  - __debug__ è normalmente True ed è False quando si usa l'opzione –O per richiedere l'ottimizzazione in fase di compilazione. Il generatore di codice non genera alcun codice per lo statement assert quando è specificata l'opzione –O .
- Non è possible assegnare valori a __debug__. Il suo valore è determinato all'inizio dell'interpretazione del codice.
- Maggiori dettagli su assert e raise in seguito

74

Assert statement

mod.py

```
class Super:
        def delegate(self):
                self.action()
        def action(self):
                assert False, 'action must be defined!'


X=Super()
X.delegate()
```

```
$ python3 -O mod.py
$ python3 mod.py
Traceback (most recent call last):
  File "mod.py", line 8, in <module>
    X.delegate()
  File "mod.py", line 3, in delegate
    self.action()
  File "mod.py", line 5, in action
    assert False, 'action must be defined!'
AssertionError: action must be defined!
```


75

Superclassi astratte

- A volte i programmatori, per rendere più evidente ciò che deve essere specificato nelle sottoclassi, usano degli **statement assert** o **degli statement raise** che lanciano l'eccezione NotImplementedError

Uso di raise nella funzione che deve essere fornita dalle sottoclassi

```
>>>class Super:
…     def delegate(self):
…             self.action()
…     def action(self):
…             raise NotImplementedError('action must be defined!')
>>> X = Super()
>>> X.delegate()
NotImplementedError: action must be defined!
```


Superclassi astratte

- L'eccezione sarà lanciata anche se il metodo delegate() è invocato su istanze di una sottoclasse di Super a meno che la sottoclasse non fornisca il metodo action() che rimpiazza quello della superclasse

```
>>> class Sub(Super): pass
>>> X = Sub()
>>> X.delegate()
NotImplementedError: action must be defined!
>>> class Sub(Super):
            def action(self): print('spam')
>>> X = Sub()
>>> X.delegate()
spam
```

Abstract Base Class (ABC)

- Rappresenta un ulteriore strumento per definire superclassi astratte
- Python, tramite il modulo **abc**, fornisce il supporto per definire formalmente una classe di base astratta

78

https://github.com/python/cpython/blob/3.6/Lib/abc.py

```
from abc import ABCMeta, abstractmethod    # need these definitions

class Sequence(metaclass=ABCMeta):
  """Our own version of collections.Sequence abstract base class."""

  @abstractmethod
  def __len__(self):
    """Return the length of the sequence."""

  @abstractmethod
  def __getitem__(self, j):
    """Return the element at index j of the sequence."""
```

Una metaclasse fornisce un modello per la definizione della classe stessa, **ABCMeta** assicura che il costruttore della classe lanci un'eccezione quando si tenta di istanziare la classe astratta

@abstractmethod è un decoratore, indica che non si fornisce un'implementazione del metodo (il metodo è astratto) e le classi derivate devono implementarlo (Python impone ciò impedendo l'istanziazione di sottoclassi che non implementano i metodi dichiarati astratti)

79

metodi comuni a tutte le sequenze

```
def __contains__(self, val):
  """Return True if val found in the sequence; False otherwise."""
  for j in range(len(self)):
    if self[j] == val:                # found match
      return True
  return False

def index(self, val):
  """Return leftmost index at which val is found (or raise ValueError)."""
  for j in range(len(self)):
    if self[j] == val:                # leftmost match
      return j
  raise ValueError('value not in sequence')    # never found a match

def count(self, val):
  """Return the number of elements equal to given value."""
  k = 0
  for j in range(len(self)):
    if self[j] == val:                # found a match
      k += 1
  return k
```

i metodi __contains__, index e count non si basano su nessuna assunzione di come è realizzata l'istanza self

80

Abstract Base Class (ABC)

- Sebbene quest'ultima tecnica per creare superclassi astratte richieda più codice e la conoscenza di strumenti più avanzati, un vantaggio di questo approccio è che gli errori che scaturiscono dall'assenza di metodi si verificano quando tentiamo di creare un'istanza della classe e non più tardi quando tentiamo di invocare il metodo mancante.

81

Programmazione avanzata a.a. 2020-21

A. De Bonis

OOP: Metodi statici e di classe

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