Programmazione avanzata a.a. 2020-21

A. De Bonis

Introduzione a Python

VIII lezione

79

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

80

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!
```


81

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 possibile assegnare valori a __debug__. Il suo valore è determinato all'inizio dell'interpretazione del codice.
- Maggiori dettagli su assert e raise in seguito

82

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!
```


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
```


85

Abstract Base Class (ABC)

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

86

```
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 la classe non 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)

87

alternativa al codice della slide precedente: deriviamo la classe dalla classe ABC

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

class Sequence(ABC):
  """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."""
```

La classe ABC ha **ABCMeta** come metaclasse

88

metodi comuni a tutte le sequenze e implementati nella classe base Sequence

```
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

89

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.

testSequence.py

```
import seq
s=seq.Sequence()
```

```
Traceback (most recent call last):
  File "/Users/adb/testSequence.py", line 4, in <module>
    s=seq.Sequence()
TypeError: Can't instantiate abstract class Sequence with abstract methods __getitem__, __len__
```


90

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 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

91

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 cls 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")

92

I metodi statici e i metodi di classe

- la funzione printNumInstances (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)
```


93

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
```


94

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**

95

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]
```


96

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
```


97

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
```


98

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
```


99

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)
```


100

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
```


101

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'>
```


102

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
```


103

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)
```


104

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])
```


105

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 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

106

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
```