Programmazione avanzata a.a. 2023-24

A. De Bonis

Introduzione a Python

VI lezione

75

Gestire l'eccezione

file exampleExceptions1.py

```
import sys

try:
      f = open(sys.argv[2], sys.argv[1])
      print(sys.argv[2], 'has', len(f.readlines()), 'lines')
      f.close()
except OSError:
      print('cannot open', sys.argv[2])
except:
   print('errore inatteso')
   raise       #viene rilanciata l'eccezione prodotta nel try
```

Se non viene lanciata l'eccezione OSError o di un tipo che è sottoclasse di OSError, ogni altro tipo di eccezione viene catturata dalla seconda clausola except

eseguiamo lo script con **python exampleExceptions1.py r filename**

se il file filename non esiste

cannot open filename

se il file filename esiste e contiene tre linee

filename has 3 lines

qui l'eccezione gestita dalla prima clausola except è di tipo FileNotFoundError

76

Gestire l'eccezione

file exampleExceptions1.py

```
import sys

try:
      f = open(sys.argv[2], sys.argv[1])
      print(sys.argv[2], 'has', len(f.readlines()), 'lines')
      f.close()
except OSError:
      print('cannot open', sys.argv[2])
except:
   print('errore inatteso')
   raise          #viene rilanciata l'eccezione prodotta nel try
```

Se non viene lanciata l'eccezione OSError, ogni altro tipo di eccezione viene catturata dalla seconda clausola except

eseguiamo lo script con **python exampleExceptions1.py y filename**

```
errore inatteso
Traceback (most recent call last):
  File "exampleExceptions1.py", line 5, in <module>
    f = open(arg, sys.argv[1])
ValueError: invalid mode: 'y'
```


77

Gestire l'eccezione

file exampleExceptions2.py

```
import sys

for arg in sys.argv[1:]:
   try:
      f = open(arg, 'r')
   except OSError:
      print('cannot open', arg)
   else:
      print(arg, 'has', len(f.readlines()), 'lines')
      f.close()
```

- In questo esempio (dalla documentazione) la modalita` di apertura del file è fissata nel codice e sulla linea di comando vengono passati i nomi dei file.
- Se non viene lanciata l'eccezione OSError, vuol dire che il file è aperto in lettura e nella clausola else viene stampato il numero di linee del file

eseguiamo lo script con **python exampleExceptions2.py file1 file2 file3**

file1 e file3 non esistono; file2 esiste e contiene 4 linee

```
cannot open file1
file2 has 4 lines
cannot open file3
```


78

Lanciare un'eccezione

- Quando si lancia un'eccezione con raise, l'eccezione puo` essere creata invocando il costruttore della classe con o senza argomenti.
  - Se non si specificano degli argomenti allora si puo` usare semplicemente il nome della classe (senza le parentesi tonde).
- Gli argomenti vengono memorizzati nella variabile args dell'eccezione istanziata
- In una clausola except il nome dell'eccezione puo` essere seguita da una variabile. In questo caso la variabile si lega all'istanza dell'eccezione con gli argomenti dell'eccezione memorizzati in .args.
- L'istanza dell'eccezione definisce __str__() in modo che gli argomenti possano essere stampati direttamente senza dover fare riferimento ad .args.

79

Lanciare un'eccezione

- Esempio presente nella documentazione:

```
>>> try:
...     raise Exception('spam', 'eggs')
... except Exception as inst:
...     print(type(inst))    # the exception instance
...     print(inst.args)     # arguments stored in .args
...     print(inst)          # __str__ allows args to be printed directly,
...                          # but may be overridden in exception subclasses
...     x, y = inst.args     # unpack args
...     print('x =', x)
...     print('y =', y)
...
<class 'Exception'>
('spam', 'eggs')
('spam', 'eggs')
x = spam
y = eggs
```


80

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**
- **next()** comincia l'esecuzione di un generatore o la riprende dall'ultima espressione yield eseguita ripartendo da dove l'esecuzione era stata sospesa

81

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**

82

Generatori

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

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

da len(data)-1 a 0 (-1 è escluso)

enoizammargorp

83

qualche dettaglio in piu` su iter()

- iter() restituisce un oggetto iteratore e puo` prendere in input uno o due argomenti
- In assenza del secondo argomento, il primo argomento deve essere una collezione che supporta l'iterazione. Nel caso la collezione non supporti il protocollo dell'iterazione viene lanciato TypeError.
- Se viene fornito anche il secondo argomento allora il primo argomento deve essere un callable. In questo caso l'iteratore restituito da iter() invoca il callable ogni volta che viene invocato il suo metodo __next__(). Il secondo argomento svolge il ruolo di sentinella e se next() restituisce un valore uguale a questo argomento viene lanciata l'eccezione StopIteration.

84

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

85

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!
```


86

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

87

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!
```


88

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!
```


89

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
```


90

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

91

```
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)

92

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

93

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

94

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__
```


Esercizio su __mro__

- Esercizio: Scrivere la classe ClasseB che ha il metodo di istanza contaVarClasse(self,t,n) che
  - prende in input un tipo t e un intero n
  - restituisce il numero di variabili di classe di tipo t delle prime n classi nella gerarchia formata dalla classe di cui self e` istanza e dalle sue superclassi, le prime n secondo l'ordine indicato in __mro__ . Se il numero di classi nella suddetta gerarchia è minore di n allora vengono considerate tutte le classi della gerarchia.

- NB: Gli attributi di una classe, di un'istanza di una classe o di un modulo sono mantenuti in un dizionario (o un altro oggetto mapping) chiamato __dict__ che è a sua volta un attributo dell'oggetto.
- vars([*object*]) restituisce il __dict__ per il modulo, la classe, l'istanza o un qualsiasi altro oggetto dotato di un attributo __dict__
- Il namespace di un modulo e` mantenuto in __dict__

```
import sys
print(sys.modules[__name__].__dict__== vars())
```

```
True
```

I metodi statici e i metodi di classe

- Un metodo di istanza di una classe 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

97

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")

98

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)
```


99

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
```


100

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**

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]
```

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
```


103

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
```


104

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
```


105

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)
```


106

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
```


107

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'>
```


108

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])
```