Corso di **Python**

Lezione 8

Le eccezioni

Editor: Davide Brunato
Scuola Internazionale Superiore di Studi Avanzati di Trieste

Perché usare le eccezioni

- Intercettare errori non previsti e trattarli correttamente
- Semplificazione e chiarezza del codice
- Propagazione errori nello stack di chiamata
- Raggruppamento per tipo di errore

Usare un linguaggio che implementa la gestione delle eccezioni, anche per gli script non banali ...

Riassunto sui blocchi

- Blocchi condizionali (*if-elif-else*)
- Blocchi iterativi (*for-else; while-else*)
- Blocchi funzionali (*def*)
- Blocchi strutture/classi (*class*)

Blocchi condizionali

```
if expression:
    statement(s)
[elif expression:
    statement(s)
...
elif expression:
    statement(s)]
[else:
    statement(s)]
```

Blocchi iterativi

```
while expression:
    statement(s)
[else:
    statement(s)]


for target in iterable:
    statement(s)
[else:
    statement(s)]
```

Blocchi funzione

def *function-name*(*parameters*)**:**

statement(s)

- *Esempio:*

```
def somma(a, b):
    return a + b
```

- Forma minimale usando espressioni lambda:

lambda *parametri: espressione*

- *Esempio:* `quadrato = lambda x: x**2`

Classi e strutture

class *classname*[*(base-classes)*]***:***

statement(s)

- *Esempio:*

```
class A(object):
    x = 0
    y = 0
```

- *Forma compatta con la funzione built-in* ***type****(), che ritorna una classe:*

class type(*name*, *base-classes*, *attributes*)

```
>>> A = type('A', (object,), dict(x=0, y=0))
```

Il blocco try/except

- È il costrutto di Python per gestire un'eccezione:

```
try:
    statement(s)
except expression [as target]:
    statement(s)
[else:
    statement(s)]
```

- In Python 2 al posto di **`as`** c'era una virgola
  - Python 2.6/2.7 accettano entrambe le sintassi
  - Python 3 accetta solo **`as`**
- Come funziona questo costrutto?
  - Nel blocco *try* c'è il codice con le istruzioni critiche da eseguire
  - Nel blocco except c'è il codice per gestire l'eccezione
  - Il blocco *else* viene eseguito se non viene generata un'eccezione nel primo blocco

Il blocco except

- L'espressione che segue except è il nome di una eccezione:

```
except KeyError:
    ...
```

- Per intercettare più eccezioni si elencano tra parentesi tonde:

```
except (IOError, KeyError):
```

- Il *target* è un oggetto che può essere richiesto in via opzionale se serve avere informazioni precise sull'evento

```
except SystemError as e:
except (NameError, AttributeError) as err:
```

Generare eccezioni

- Per generare/sollevare eccezioni in modo esplicito si usa l'istruzione **raise**
- La sintassi di questa istruzione è stata modificata in Python 3:
  - Python 2: **`raise`** [*`expression`* [**`,`** *`expression`*]]
  - Python 3: **`raise`** [*`expression`* [**`from`** *`expression`*]]
- La prima espressione è l'oggetto dell'eccezione, che dev'essere sottoclasse di BaseException
- La seconda espressione è il valore
- Si può usare un'istanza della classe di eccezione per esplicitare eccezione e valore in una sola espressione (in questo caso il valore è l'istanza stessa)
- Un raise semplice senza espressioni rilancia l'ultima eccezione attiva nell'ambito
  - Se non ci sono eccezioni attive nell'ambito allora genera un errore *TypeError* *(RuntimeError* in Python3)
- La clausola *from* introdotta in Python 3 permette di concatenare l'eccezione generata con un'altra, anche la seconda può essere in forma di classe o di istanza

Except multipli

- Sono ammessi più blocchi except
- Esempio (tratto dalla documentazione ufficiale di Python):

```
import sys
try:
    f = open('myfile.txt')
    s = f.readline()
    i = int(s.strip())
except IOError as e:
    print "I/O error({0}): {1}".format(e.errno, e.strerror)
except ValueError:
    print "Could not convert data to an integer."
except:
    print "Unexpected error:", sys.exc_info()[0]
    raise
```

- L'ultimo `except` può omettere la specifica dell'eccezione: ha il significato di una *wildcard* ma è da usare con molta cautela e solo con un rilancio dell'eccezione (istruzione `raise`)

Blocco critico

- Mettere nella clausola *try* solo il blocco critico di istruzioni da proteggere
- Istruzioni successive possono andare tranquillamente nel blocco else
- Esempio:

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

Il costrutto try/finally

- È prevista una clausola *finally* per il clean-up del codice

```
try:
    statement(s)
finally:
    statement(s)
```

- Il blocco *finally* viene eseguito comunque
- Se il blocco *try* genera un'eccezione prima viene eseguito il blocco finally e poi l'eccezione viene rilanciata (*re-raised*)

try/except/else/finally

- Da Python 2.5 try/except e try/finally sono a tutti gli effetti unificati:

```
try:
    statement(s)
except expression as target:
    statement(s)
[else:
    statement(s)]
[finally:
    statement(s)]
```

- Il blocco finally viene sempre eseguito, sia che si generi un'eccezione gestita nel blocco o gestita esternamente, sia che l'esecuzione vada liscia e non venga generata nessuna eccezione

Esempio di try/except/else/finally

```
def divide(x, y):
    try:
        res = x / y
    except ZeroDivisionError:
        print("div by zero!")
    else:
        print("result is", res)
    finally:
        print("exec finally")
```

```
>>> divide(2, 1)
result is 2
executing finally clause
>>> divide(2, 0)
div by zero!
executing finally clause
>>> divide("2", "1")
executing finally clause
Traceback (most recent call
last):
File "<stdin>", line 1, in ?
File "<stdin>", line 3, in divide
TypeError: unsupported operand
type(s) for /: 'str' and 'str'
```

Propagazione delle eccezioni

- Se l'eccezione non viene intercettata dagli except del blocco allora viene propagata
- La propagazione avviene lungo lo stack di chiamata, fino a quando non c'è un blocco except attivo adatto
- Se non c'è il blocco l'eccezione arriva all'interprete che ritorna l'eccezione a livello di console

Eccezioni built-in

- **BaseException** è la classe base di tutte le eccezioni
- Gerarchia eccezioni di Python 2.7
- Possiamo derivare da **Exception** per definire nuove eccezioni
- Sottoclassi di eccezioni:
  - *StopIteration*
  - StandardError
  - Warning
- Buona pratica è creare una sottoclasse (es. MyappException) e poi derivare ulteriormente

```
BaseException
 +-- SystemExit
 +-- KeyboardInterrupt
 +-- GeneratorExit
 +-- Exception
      +-- StopIteration
      +-- StandardError
      |    +-- BufferError
      |    +-- ArithmeticError
      |    |    +-- FloatingPointError
      |    |    +-- OverflowError
      |    |    +-- ZeroDivisionError
      |    +-- AssertionError
      |    +-- AttributeError
      |    +-- EnvironmentError
      |    |    +-- IOError
      |    |    +-- OSError
      |    +-- EOFError
      |    +-- ImportError
      |    +-- LookupError
      |    |    +-- IndexError
      |    |    +-- KeyError
      |    +-- MemoryError
      |    +-- NameError
      |    |    +-- UnboundLocalError
      |    +-- ReferenceError
      |    +-- RuntimeError
      |    |    +-- NotImplementedError
      |    +-- SyntaxError
      |    |    +-- IndentationError
      |    |         +-- TabError
      |    +-- SystemError
      |    +-- TypeError
      |    +-- ValueError
      |         +-- UnicodeError
      |              +-- UnicodeDecodeError
      |              +-- UnicodeEncodeError
      |              +-- UnicodeTranslateError
      +-- Warning
           +-- DeprecationWarning
           +-- PendingDeprecationWarning
           +-- RuntimeWarning
           +-- SyntaxWarning
           +-- UserWarning
           +-- FutureWarning
           +-- ImportWarning
           +-- UnicodeWarning
           +-- BytesWarning
```

Console ed eccezioni

- La gestione delle eccezioni è integrata nella console di Python:

```
>>> a = 1/0
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: integer division or modulo by zero


>>> try:
...     a = 1/0
... except ZeroDivisionError:
...     print("Divisione per zero!")
...
Divisione per zero!
```

Errori sintattici

- Gli errori sintattici vengono intercettati immediatamente dall'interprete, anche se inseriti in una sezione *try*
- Eccezioni per errori sintattici:

```
SyntaxError
    +-- IndentationError
        +-- TabError
```

- Esempio:

```
>>> try:
...     a = 0xFA + '10'   # TypeError: Non valutata subito!
...     b = 0xFG          # SyntaxError: valutata subito!
  File "<stdin>", line 3
    b = 0xFG
          ^
SyntaxError: invalid syntax
```

Errori di identificatore

- Quando si usa un nome non esistente viene generata l'eccezione `NameError`:

```
>>> elenco
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'elenco' is not defined
```

- Ricordarsi che Python è dinamico:

```
>>> if True:
...     a = 10
...
>>> print(a)
10
>>> if False:
...     b = 20
...
>>> print(b)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'b' is not defined
```

Eccezioni di *warning*

- Se l'eccezione è un *warning* per default l'esecuzione non viene interrotta
  - `Warning`

```
+-- DeprecationWarning
+-- PendingDeprecationWarning
+-- RuntimeWarning
+-- SyntaxWarning
+-- UserWarning
+-- FutureWarning
+-- ImportWarning
+-- UnicodeWarning
+-- BytesWarning
```

- Per default alcune di questi warning generano un messaggio di avviso, altri no
- Si può impostare un comportamento diverso usando la libreria `warnings`:

```
>>> import warnings
>>> warnings.filterwarnings("always", category=DeprecationWarning)
>>> warnings.filterwarnings("error", category=DeprecationWarning)
>>> warnings.resetwarnings()
```

Eccezione **TypeError**

- Viene generata quando il tipo dell'argomento è inappropriato
- Esempi:

```
>>> abs('10')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: bad operand type for abs(): 'str'


>>> res = 0xAA + 'FF'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for +:
'int' and 'str'
```

Eccezione **ValueError**

- Viene generata quando il tipo dell'argomento è corretto ma il valore è inappropriato
- Esempi:

```
>>> int('49')
49
>>> int('FF', base=16)
255
>>> int('FF')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: invalid literal for int() with base
10: 'FF'
```

Eccezione **KeyError**

- Errore generato quando si accede ad un dizionario con chiave non esistente
- Esempio:

```
>>> emails = {'Tizio': 'tizio@example.com', 'Caio': 'caio@example.com'}
>>> emails['Sempronio']
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'Sempronio'
```

- Spesso sui dizionari è preferito il try-except al posto del controllo condizionale preventivo:

```
try:
    valore = emails[nome]
except KeyError:
    valore = None

if nome in emails:
    valore = emails[nome]
else:
    valore = None
```

Eccezione **IndexError**

- Errore generato quando si accede ad una sequenza con un indice fuori dai limiti:

```
>>> numeri = (10, 20, 30, 40, 50)
>>> numeri[0]
10
>>> numeri[4]
50
>>> numeri[5]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: tuple index out of range
>>> numeri[-1]
50
>>> numeri[-10]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: tuple index out of range
```

Eccezione **AttributeError**

- Errore generato quando si accede ad un attributo inesistente per l'oggetto:

```
>>> class A(object):
...     x = 0
...
>>> A.x
0
>>> A.y
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: type object 'A' has no attribute 'y'
>>> A.y = 0
>>> A.y
0
```

Eccezioni e moduli

- Le eccezioni risultano molto utili anche in fase di importazione di moduli
- In caso di errore viene generata l'eccezione *ImportError*
- Ad esempio in questo caso si vuole importare un modulo che ha cambiato nome nella versione 3.x di Python:

```
try:
    import configparser
except ImportError:
    # Fall back for Python 2.x
    import ConfigParser as configparser
```

Definire nuove eccezioni

- Due esempi di eccezioni tratti da un programma, che definiscono specifici messaggi di errore:

```
class FileAccessError(Exception):
    def __init__(self, message):
        Exception.__init__(self, message)
        logger.debug('!FileAccessError: {0}'.format(message))


class OptionError(Exception):
    def __init__(self, option, message=None):
        if message is None:
            message = 'syntax error for option "{0}"'.format(option)
        else:
            message = 'option "{0}": {1}'.format(option, message)
        Exception.__init__(self, message)
        logger.debug('!OptionError: {0}'.format(message))
```

- Ma si può anche decidere di creare nuove eccezioni, senza modifiche interne, allo scopo di creare dei blocchi except specifici nel nostro codice:

```
class MyException(Exception):
    pass
```

Contesti ed eccezioni

- Si usa l'istruzione **with**, che definisce un contesto di esecuzione**:**

```
with expression[ as target]:
    statement(s)
```

- Il contesto di esecuzione garantisce un clean-up predefinito in caso di eccezione
- Esempio:

```
for line in open('myfile.txt'):
    print(line)

with open('myfile.txt') as f:
    for line in f:
        print(line)
```

Nel secondo caso il file viene chiuso automaticamente, anche in caso Il codice interno al contesto generi un'eccezione