

# Gestione degli errori e Sequenze di Dati

Informatica@SEFA 2018/2019 - Lezione 8

**Massimo Lauria** <massimo.lauria@uniroma1.it>  
<http://massimolauria.net/courses/infosefa2018/>

**Venerdì, 12 Ottobre 2018**

# Gestione degli errori

# Segnalare un errore

In caso di errore l'esecuzione del programma si interrompe ed python vi comunica cosa è andato storto.

```
5 / 0
```

1

```
Traceback (most recent call last):  
  File "<stdin>", line 1, in <module>  
ZeroDivisionError: division by zero
```

```
2+"ciao"
```

1

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

# Causare un errore volontariamente

Una funzione può causare volontariamente una situazione di errore per segnalare, ad esempio, che

- ▶ i parametri passati non vanno bene
- ▶ delle operazioni non sono andate a buon fine

# Sintassi

```
raise NomeDellErrore 1
raise NomeDellErrore("Messaggio opzionale") 2
raise NomeDellErrore("Messaggio opzionale") 3
```

```
def scontato(prezzo, sconto): 1
    if not ( 0 <= sconto <= 100 ): 2
        raise ValueError("Lo sconto non è tra zero e cento.") 3
    4
    return prezzo*(1.0 - sconto/100.0 ) 5
6
print( scontato(500, 20) ) 7
print( scontato(500,-10) ) 8
```

400.0

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

ValueError: Lo sconto deve essere è tra zero e cento.

# Significato di un errore

Sollevare un errore in una funzione

- interrompe il programma
- l'errore viene segnalato all'utente

Esiste la possibilità di far gestire al programma stesso gli errori che produce. Noi non vedremo questa cosa.

# Alcuni tipi di errore

- ▶ `TypeError` un valore ha tipo sbagliato.
- ▶ `ValueError` valori in passato non sono accettabili
- ▶ `NameError` nome di variabile o funzione non esiste
- ▶ `ZeroDivisionError` divisione per zero

e molti altri.

# Ancora con scontato (I)

```
from numbers import Real          1
# Real è un tipo di dati che "comprende" int e float          2
                                                                    3
def scontato(prezzo,sconto):      4
    if not isinstance(prezzo, (int,float) ):                    5
        raise TypeError("prezzo deve essere di tipo numerico")6
                                                                    7
    if not isinstance(sconto, Real ):                            8
        raise TypeError("sconto deve essere di tipo numerico")9
                                                                    10
    if not ( 0 <= sconto <= 100 ):                             11
        raise ValueError("Lo sconto non è tra zero e cento.")12
                                                                    13
    return prezzo*(1.0 - sconto/100.0 )                       14
```



# Ancora con scontato (II)

```
print( scontato(500,20) )  
print( scontato(500,'ciao') )
```

1  
2

400.0

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

raise TypeError("prezzo e sconto devono essere tipi numerici")

TypeError: prezzo e sconto devono essere tipi numerici

# Type checking e isinstance

```
isinstance( expr, tipo )
```

1

restituisce

- ▶ True se **expr** ha valore del tipo richiesto
- ▶ False altrimenti

```
isinstance( expr, (tipo1, tipo2, ... ) )
```

1

restituisce

- ▶ True se il tipo di **expr** è uno tra (tipo1, tipo2, ...)
- ▶ False altrimenti

# Input da tastiera

# Leggere dati da tastiera

La funzione `input` permette di inserire dei dati da tastiera.

```
>>> x = input()
jdhasjkj
>>> type(x)
<class 'str'>
>>> x
'jdhasjkj'
>>>
```

# Dati non testuali

Quello che viene letto è sempre una stringa. Se si vuole un dato diverso si deve effettuare una conversione.

```
>>> x = input()
42
>>> type(x)
<class 'str'>
>>> x
'42'
>>> numero = int(x)
>>> numero
42
>>> type(numero)
<class 'int'>
```

# Aggiungere un testo esplicativo

È possibile far stampare ad `input()` un testo prima che il python si metta ad aspettare l'input da tastiera.

```
>>> x = input("INPUT> ")  
INPUT>
```

# Verificare sempre i dati in input

La maggior parte dei problemi di sicurezza di un software deriva dal non verificare adeguatamente i dati in input.

```
>>> x = input("Inserisci un numero per favore: ")
Inserisci un numero per favore: 43234wsd
>>> x
'43234wsd'
>>> int(x)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: invalid literal for int() with base 10: '43234wsd'
>>>
```

Controllate e pulite **sempre** i dati letti.

# Sequenze di valori

\* molte delle slide di questa sezione non sono altro che riproduzioni del codice del capitolo 6 del libro.



# Moltitudini di valori

Le variabili che abbiamo visto fino ad ora contengono solo un valore.

È possibile mantenere moltitudini di valori

- logicamente raggruppati
- operandovi collettivamente

La forma più semplice sono le **sequenze** di valori

# Sequenze

Una sequenza in Python è un tipo di dato che contiene una serie finita di valori

$$v_0 \quad v_1 \quad v_2 \quad \dots \quad v_n$$

Tali che si possa accedere al valore di ognuna delle posizioni.

# Le stringhe sono sequenze di caratteri

```
stringa = 'la rana in spagna gracida in campagna'      1
print( len( stringa) )      # quanti caratteri          2
                                                                3
print ( stringa[0] )      4
print ( stringa[10])      5
print ( stringa[12])      6
print ( stringa[36])      7
```

37

l

p

a

- ▶ `len(s)` fornisce la lunghezza della stringa `s`
- ▶ `s[i]` fornisce il valore dell'`i`-esimo carattere.

# Liste : le sequenze più importanti e flessibili

```
primi = [2, 3, 5, 7, 11, 13, 17, 19]      1
print(primi)                              2
                                           3
colori = ['blu', 'rosso', 'verde', 'giallo'] 4
print(colori)                             5
                                           6
misc = ['red', 2, 3.14, 'blue']           7
print(misc)                              8
                                           9
lista_vuota = []                          10
print(lista_vuota)                       11
                                           12
print(type(colori))                      13
```

```
[2, 3, 5, 7, 11, 13, 17, 19]
['blu', 'rosso', 'verde', 'giallo']
['red', 2, 3.14, 'blue']
[]
<class 'list'>
```

# Tuple: gruppi di valori

```
tupla = ('ciao', 1, 2)
```

```
print(tupla)
```

```
tupla_vuota = ()
```

```
print(tupla_vuota)
```

```
print(type( (1,2) ))
```

1

2

3

4

5

6

7

```
('ciao', 1, 2)
```

```
()
```

```
<class 'tuple'>
```

# Abbiamo già incontrato le tuple

```
def quoziente_e_resto(N,D):           1
    if D==0:                           2
        return None    # pessima gestione dell'errore  3
                                        4
    quoziente = N // D                 5
    resto = N % D                     6
                                        7
    return (quoziente, resto)         8
                                        9
print( quoziente_e_resto(10,4) )      10
print( quoziente_e_resto(7, 2) )      11
```

```
(2, 2)
(3, 1)
```

# Sintassi di liste vs tuple

```
[ expr1, expr2, expr3, expr4, expr5]      # lista      1
```

```
( expr1, expr2, expr3, expr4, expr5 )    # tupla      1
```

Le parentesi tonde possono essere omesse negli assegnamenti e nei `return`, ma è sconsigliabile.

```
t = 'Aa', 'Bb', 'Cc'                      1
                                           2
def dummy():                               3
    return 1,2,3,4                          4
                                           5
print( t )                                 6
print( dummy() )                           7
```

```
('Aa', 'Bb', 'Cc')
(1, 2, 3, 4)
```

# Operazioni di base

- ▶ `len(seq)` indica il numero di valori nella sequenza
- ▶ `seq[i]` fornisce il valore dell'*i*-esimo elemento

```
tupla = ('ciao', 1, 2) 1
lista = ['red', 2, 3.14, 'blue'] 2
3
print("Lunghezza della tupla:", len( tupla) ) 4
print("Lunghezza della lista:", len( lista) ) 5
6
print ( tupla[0] ) 7
print ( tupla[1] ) 8
9
print ( lista[1] ) 10
print ( lista[2] ) 11
```

```
Lunghezza della tupla: 3
Lunghezza della lista: 4
ciao
1
2
3.14
```



# “Gli informatici contano da 0”

L'ultimo valore in `seq` ha indice `len(seq)-1`

```
seq = ['ciao', 2, 1.2 , "blah"]      1
                                     2
print(seq[0])                        3
print(seq[1])                        4
print(seq[2])                        5
print(seq[3])                        6
print(seq[4])    # errore!           7
```

```
ciao
2
1.2
blah
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
    print(seq[4])    # errore!
IndexError: list index out of range
```

# IndexError

In `seq[i]`, l'espressione `i` è l'**indice** dell'elemento in posizione `i`-esima.

Se `i` è **maggiore o uguale** di `len(seq)` allora Python solleva un `IndexError`.

# Tipi di sequenze importanti

- ▶ tuple
- ▶ liste
- ▶ stringhe
- ▶ intervalli (`range`)
- ▶ ...

(li vedremo più avanti)

# Liste vs Tuple : mutabilità

Finora liste e tuple sembrano identiche ma hanno una differenza **fondamentale**

- ▶ Le tuple non possono essere modificate
- ▶ Le liste posso cambiare lunghezza e valori memorizzati.

# Modifiche ai valori della lista

L'elemento  $i$ -esimo della lista  $L$  può essere assegnato come fosse una variabile.

```
L = [2, 'sardegna' , 'piemonte', 4.5, 'lazio']      1
print(L)                                           2
                                                    3
L[2] = 'molise'                                    4
print(L)                                           5
```

```
[2, 'sardegna', 'piemonte', 4.5, 'lazio']
[2, 'sardegna', 'molise', 4.5, 'lazio']
```

# Aggiunta di un elemento alla fine della lista

La lista `L` può essere “allungata” di un elemento

```
L = ['sardegna' , 'piemonte', 'lazio']      1
print(L)                                    2
print(len(L))                               3
                                           4
L.append('molise')                           5
print(L)                                    6
print(len(L))                               7
                                           8
L.append('sicilia')                          9
print(L)                                    10
print(len(L))                               11
```

```
['sardegna', 'piemonte', 'lazio']
3
['sardegna', 'piemonte', 'lazio', 'molise']
4
['sardegna', 'piemonte', 'lazio', 'molise', 'sicilia']
5
```

# Eliminazione di elementi in una lista

## Operatore del

```
L = ['sardegna', 'piemonte', 'lazio', 'molise', 'sicilia'] 1
print(L) 2
print(len(L)) 3
4
# elimina il 4 elemento, e trasla 5
# a sinistra i successivi 6
del L[3] 7
print(L) 8
9
print( L[2] , L[3]) 10
```

```
['sardegna', 'piemonte', 'lazio', 'molise', 'sicilia']
5
['sardegna', 'piemonte', 'lazio', 'sicilia']
lazio sicilia
```

# Uso di Tuple vs Liste

## Liste

- ▶ una serie di dati omogenei
- ▶ serie temporale
- ▶ una lista di osservazioni
- ▶ dati da aggiornare

## Tupla

- ▶ un singolo dato multidimensionale
- ▶ una voce in una rubrica telefonica
- ▶ un punto cartesiano  $(x, y)$
- ▶ un colore RGB  $(r, g, b)$



# Entità immutabili

Un'entità **immutabile** non può essere modificata dopo la sua definizione

## Mutabili

- ▶ **liste** `<class 'list'>`

## Immutabili

- ▶ **stringhe** `<class 'str'>`
- ▶ **tuple** `<class 'tuple'>` (quasi immutabile)
- ▶ **intervalli** `<class 'range'>`

# Piccoli dettagli sulla tupla “quasi immutabile”

Se una tupla contiene una lista, e il valore di quella lista cambia, allora anche il valore della tupla.

<code>tupla = (1,2,[3,4])</code>	1
<code>print(tupla)</code>	2
	3
<code>tupla[2][0] = "mod"</code>	4
<code>print(tupla)</code>	5

```
(1, 2, [3, 4])  
(1, 2, ['mod', 4])
```

- ▶ Non si può mettere un altro oggetto in posizione 2
- ▶ Però l'oggetto è una lista, e il **suo valore** può cambiare.

# Lecture

- ▶ Capitolo 6