INDICE

gestione errori con cicli if

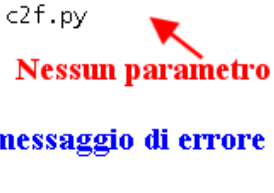
consideriamo lo script c2f.py della lezione sulla gestione dei parametri da riga di comando. che riportiamo per intero qui sotto.

""" esempio di gestione di un paragrafo da riga di comando con sys.argv """ def c2f(C): return 9.*C/5+32. if __name__=='__main__': import sys C = float(sys.argv[1]) print(c2f(C))

vediamo cosa succede se invoco lo script senza pasargli parametri:

```
C:\python\corso\materialexcorsopyton\work>python c2f.py
Traceback (most recent call last):
  File "c2f.py", line 11, in <module>
    C = float(sys.argv[1])
IndexError: list index out of range

C:\python\corso\materialexcorsopyton\work>
```



Il messaggio di errore è significativo ed in particolare ci dice:

- il modulo/script in cui c'e' l'espressione/statement che ha provocato l'errore.
- il numero di riga del codice sorgente e la riga stessa
- il tipo di errore (IndexError)

Analogamente, se invoco lo script passando come parametro un carattere o una stringa ...

```
C:\python\corso\materialexcorsopyton\work>python c2f.py ciao
Traceback (most recent call last):
  File "c2f.py", line 11, in <module>
    C = float(sys.argv[1])
ValueError: could not convert string to float: 'ciao'
```

... l'interprete ci segnala un `ValueError`

Potrei pensare di gestire la situazione con una serie di cicli if.
ad esempio per gestire l'index error potrei eseguire un test simile:

In [ ]:
```
"""
c2f_if.py
esempio di gestione di un paragrafo da riga di comando
con sys.argv e gestione index error con ciclo if
"""

def c2f(C):
    return 9.*C/5+32.

if __name__=='__main__':
    import sys

    #Aggiungo questo!!!
    if len(sys.argv) < 2 :                        # testo se c'e' almeno un parametro
        print('specificare almeno un parametro') # se non c'e' stampo un messaggio di errore
        sys.exit                                  # ed esco
```

```
    #fine aggiunta

    C = float(sys.argv[1])
    print(c2f(C))
```

gestire la situazione che porta al ValueError implica un test sul tipo di dato che deve essere compatibile con la conversione a float

In [ ]:
```
"""
c2f_if_2.py
esempio di gestione di un paragrafo da riga di comando
con sys.argv e gestione index error con ciclo if
"""

def c2f(C):
    return 9.*C/5+32.

if __name__=='__main__':
    import sys

    #Aggiungo questo!!!
    if len(sys.argv) < 2 :                        # testo se c'e' almeno un parametro
        print('specificare almeno un parametro') # se non c'e' stampo un messaggio di errore
        sys.exit(1)                               # ed esco
    #fine aggiunta

    if not sys.argv[1].isnumeric(): #funziona solo con gli interi ...
        print('parametro di comando non numerico')
        sys.exit(1)


    C = float(sys.argv[1])
    print(c2f(C))
```

Per una gestione soddisfacente dei parametri passati a float dovremmo prevedere sia il separatore decimale sia un numero passato in notazione scientifica e la questione potrebbe farsi complicata.

Sulla base del motto *'è più facile chiedere scusa che chiedere il permesso'* sono state introdotte le `eccezioni`

Le eccezioni

C'è un altro modo per gestire gli errori, che è indicato come `metodo standard` ed è preferibile specialmente nel caso di programmi articolati nei quali la gestione di un errore puo'/deve essere effettuata in porzioni di codice distinte da quella in cui l'errore ha avuto origine (cosa non possibile con semplici cicli if).

Come si è già detto, le eccezioni fanno riferimento al paradigma di programmazione noto come EAFP che sta per `Easier to ask for forgiveness than permission`.
Cioe' piuttosto che cercare di immaginarsi tutte le condizioni che possono dare luogo ad un errore si gestisce l'errore una volta che questo si è manifestato.

Quando si manifesta un errore il normale flusso del programma si interrompe ed il controllo passa al meccanismo di gestione delle eccezioni che cerca il gestore più appropriato per l'errore che si è manifestato.

try: ... except:

la sintassi del costrutto base per la gestione delle eccezioni è la seguente:

try: except:

c2f con gestione errore generico

Nella forma di base, quella con except e nessun tipo di errore specificato, qualsiasi errore scaturisca dalle istruzioni dopo try viene gestito dalle istruzioni indentate dopo except:

In [ ]:
```
"""
c2f_eccezioni_00.py
esempio di gestione di un paragrafo da riga di comando
con sys.argv e gestione errori con le eccezioni
```

```
00 - gestione non diversificata per tipo di errore
"""

def c2f(C):
    "funzione di conversione da Celsius a Fahrenheit"
    return 9.*C/5+32.

if __name__=='__main__':
    import sys

    try:
        C = float(sys.argv[1])
    except:
        print("Errore nel passaggio della temperatura C da riga di comando")#messaggio di
errore
        sys.exit(1)#esco

    print('{0} C sono {1} F'.format(C,c2f(C)))
```

vediamo cosa succede se non passo alcun parametro o se passo una stringa:

```
C:\python\corso\materialexcorsopyton\work>python c2f_eccezioni_00.py ciao
Errore nel passaggio della temperatura C da riga di comando

C:\python\corso\materialexcorsopyton\work>python c2f_eccezioni_00.py
Errore nel passaggio della temperatura C da riga di comando

C:\python\corso\materialexcorsopyton\work>python c2f_eccezioni_00.py 2
2.0 C sono 35.6 F
```

la forma `try: ... except: ...` è relativamente poco utilizzata perché gestisce qualsiasi tipo di errore e non permettere una differenziazione di gestione in base al tipo di problema che si verifica.

try: ... except tipoerrore: ...

In realtà il costrutto try ... except e' piu' articolato e permette di distinguere per tipologia di errore
La sintassi da utilizzare è la seguente:

try: except : except : ... ... else: finally:

vediamo come e' possibile modificare ulteriormente lo script c2f ...

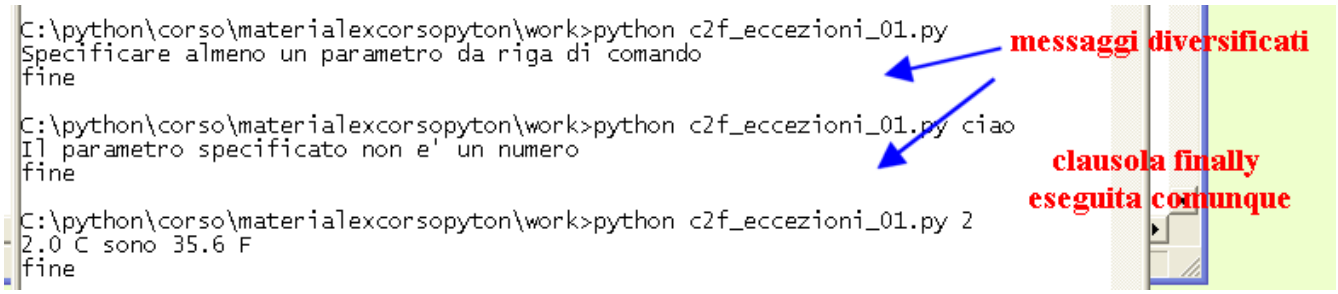
In [ ]:

```
"""
c2f_eccezioni_01.py
esempio di gestione di un paragrafo da riga di comando
con sys.argv e gestione errori con le eccezioni
01 - gestione diversificata per tipo di errore e blocchi else e finally
"""

def c2f(C):
    "funzione di conversione da Celsius a Fahrenheit"
    return 9.*C/5+32.

if __name__=='__main__':
    import sys
    try:
        C = float(sys.argv[1])
    except IndexError:
        print("Specificare almeno un parametro da riga di comando")
    except ValueError:
        print("Il parametro specificato non e' un numero")
    else:
        print('{0} C sono {1} F'.format(C,c2f(C)))
    finally:
        print('fine')
```

vediamo il risultato:



vediamo che tipi di errore definisce Python (se volete provare da shell interattivo usare `builtins` e non `builtin`)

In [4]:
```
for s in dir(__builtin__):
        if s.find('Error')>=0:
                print(s)
```

```
ArithmeticError
AssertionError
AttributeError
BufferError
EOFError
EnvironmentError
FloatingPointError
IOError
ImportError
IndentationError
IndexError
KeyError
LookupError
MemoryError
NameError
NotImplementedError
OSError
OverflowError
ReferenceError
RuntimeError
SyntaxError
SystemError
TabError
TypeError
UnboundLocalError
UnicodeDecodeError
UnicodeEncodeError
UnicodeError
UnicodeTranslateError
ValueError
WindowsError
ZeroDivisionError
```

- Le eccezioni in Python sono realizzate come una `gerarchia di classi`
- Se io raccolgo con except una eccezione di un tipo specificato (ad esempio ArithmeticError) in realtà raccoglierò tutte le eccezioni derivate da essa (ZeroDivisionError)

esempio

In [44]:
```
def dividi(a,b):
    return a/b

def usadividi(a,b):
    try:
        q = dividi(a,b)
    except ZeroDivisionError:
        print("errore in dividi({0},{1}): b deve essere diverso da zero".format(a,b))
    except TypeError:
        print("errore in dividi({0},{1}): tipi di operandi non compatibili con la
divisione".format(a,b))
```

```
    else:
        print('dividi({0},{1}) = {2}'.format(a,b,q))


usadividi(2,2)
usadividi(2,0)
usadividi('ciao',1)
```

```
errore in dividi(2,0): b deve essere diverso da zero
dividi(2,2) = 1.0
errore in dividi(ciao,1): tipi di operandi non compatibili con la divisione
```

- Il precedente esempio illustra la delocalizzazione della gestione di un errore mediante eccezioni.
- In altre parole l'errore è gestito in una funzione esterna rispetto a quella in cui c'e' l'espressione che fa scaturire l'errore.
- è da notare che in questo caso non è stato necessario intervenire sul codice della funzione dividi!!
- ciò è particolarmente significativo nel caso di funzioni molto complesse ed articolate ...

esercizio

consideriamo la formula già utilizzata in un esempio precedente, che esprime la posizione di un punto uniformemente accelerato in funzione dell'istante t, dell'accelerazione a e delle condizioni iniziali v0 e s0 per t=0:

In [ ]:
```
def posiz(t,a,v0,s0):
    return s0 + v0*t + 0.5*a*t**2
```

creare uno script `posiz_eccezioni_00.py` che:

- gestisce i parametri da riga di comando usando sys.argv, fissandone l'ordine
- non definisce alcun parametro di default e gestisce il caso di argomento mancante
- usa le eccezioni per gestire il caso di parametro non numerico
- se tutto va bene restituisce una stringa ('posiz(t={0} , a={1} , v0={2} , s0={3}) = {4}'.format(t,a,v0,s0,posiz(t,a,v0,s0)))
- prima di uscire stampa comunque la stringa fine

esercizio

creare uno script `posiz_eccezioni_01.py` che:

- gestisce i parametri da riga di comando usando il modulo getopt
- definisce dei valori di default per i quattro parametri t,v,a,s0
- usa le eccezioni per gestire il caso di parametro non numerico
- usa le eccezioni per gestire il caso di parametro opzione sconosciuta e nel caso stampa a video un messaggio in cui si riassumono le modalità di utilizzo del programma (vedi immagine sotto)
- se tutto va bene restituisce una stringa ('posiz(t={0} , a={1} , v0={2} , s0={3}) = {4}'.format(t,a,v0,s0,posiz(t,a,v0,s0)))
- prima di uscire stampa comunque la stringa fine

```
C:\python\corso\lezioni\esercizi\dispense>python posiz_eccezioni_01.py -XX
Opzione -X non gestita
Opzioni valide:
 -T <tempo in secondi>
 -S <posizione iniziale in metri>
 -V <velocita' iniziale in metri/secondo>
 -A <accelerazione in metri/secondo**2>
fine

C:\python\corso\lezioni\esercizi\dispense>
```

Gerarchia delle eccezioni

NB: le gerarchia di classi nell'immagine sottostante non è aggiornata

```
Exception
    StandardError
        Arithmetic Error
            FloatingPointError
            OverflowError
            ZeroDivisionError
        AssertionError
        AttributeError
        EnvironmentError
            IOError
            OSError
        EOFError
        ImportError
        KeyBoardInterrupt
        LookupError
            IndexError
            KeyError
        MemoryError
        NameError
        RuntimeError
        SyntaxError
        SystemExit
        TypeError
        ValueError
```

con il modulo inspect verifichiamo che la figura sopra si riferisce ad una precedente versione di Python...

In [1]:
```
import inspect
inspect.getmro(ZeroDivisionError)
```

Out[1]:
```
(builtins.ZeroDivisionError,
 builtins.ArithmeticError,
 builtins.Exception,
 builtins.BaseException,
 builtins.object)
```

sollevare un'eccezione (raise)

Il comando raise puo' essere utilizzato i due modi

sollevare un'eccezione di tipo specifico

posso sollevare una eccezione specifica con il comando raise seguito dall'eccezione:

In [2]:
```
raise IndexError('IndexError generato con raise')
```

```
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
<ipython-input-2-2814c2e16525> in <module>()
----> 1 raise IndexError('IndexError generato con raise')

IndexError: IndexError generato con raise
```

passare il controllo di un'eccezione sollevata da altri

usando `raise` senza fargli seguire una eccezione specifica permette di non interrompere la propagazione di un'altra eccezione eventualmente intercettata.

nel seguente esempio si usano entrambe le forme di raise

In [5]:
```
def generaIndexError(propaga):
    try:
        #genero l'eccezione
        raise IndexError('index error')
    except IndexError:#intercetto l'eccezione localmente
        if not propaga:
            print("raccolgo localmente ma non propago l'eccezione")
        else:
            print("raccolgo localmente e propago l'eccezione")
            raise #propago l'eccezione
```

propago l'eccezione

In [6]:
```
try:
    generaIndexError(propaga=True)
except:
    print("raccolgo globalmente l'eccezione")
```

```
raccolgo localmente e propago l'eccezione
raccolgo globalmente l'eccezione
```

non propago l'eccezione

In [7]:
```
try:
    generaIndexError(propaga=False)
except:
    print("raccolgo globalmente l'eccezione")
```

```
raccolgo localmente ma non propago l'eccezione
```

Uso degli attributi di un'eccezione

- a volte è necessario interagire direttamente con l'entità eccezione.
- e' possibile farlo associandogli un nome con `as` quando la si raccoglie con `except`
- in tal modo è possibile utilizzare eventuali attributi associati all'eccezione nel momento in cui la si è sollevata.

In [10]:
```
try:
    #genero un'eccezione passandogli come argomento un dizionario
    dictinfo={'autore':'Nicola','contesto':'lezione 09'}
    myex = IndexError(dictinfo)
    raise myex
except IndexError as err: #raccolgo l'eccezione e gli associo un nome
    #uso gli attributi della specifica eccezione raccolta
    print('dir(err)=',dir(err))
    print('args[0]=')
    for k,v in err.args[0].items():
        print(k,' : ',v)
```

```
dir(err)= ['__cause__', '__class__', '__context__', '__delattr__', '__dict__', '__doc__', '__eq__',
'__format__', '__ge__', '__getattribute__', '__gt__', '__hash__', '__init__', '__le__', '__lt__',
'__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__', '__setattr__', '__setstate__',
'__sizeof__', '__str__', '__subclasshook__', '__traceback__', 'args', 'with_traceback']
args[0]=
autore  :  Nicola
contesto  :  lezione 09
```

assert

L'istruzione assert si usa in un contesto un po'diverso dalle eccezioni tanto è che viene trascurata dall'interprete quando si usa python in modalità ottimizzata.

Mentre le eccezioni si usano per la gestione degli errori in qualche modo 'attesi', le asserzioni si usano come metodo 'difensivo' in grado di evidenziare situazioni potenzialmente pericolose in un determinato punto del programma.

Le asserzioni non si riferiscono ad uno specifico insieme di istruzioni (come quelle nel blocco try di un costrutto per la gestione delle eccezioni).

Piuttosto le asserzioni si riferiscono ad una condizione che può (ma non dovrebbe) manifestarsi in uno specifico punto del programma.

La sintassi del costrutto assert è la seguente:

assert condizione[, espressione]

Nel caso la condizione dia luogo a False allora viene sollevato un *AssertionError* che ha per eventuale argomento l'espressione opzionale.

vediamo un esempio:

In [7]:
```
#....
#x è una variabile che rappresenta una quantità che deve essere positiva
#altrimenti qualcosa è andato storto in precedenza e potrebbe andare storto in futuro

x = -1

assert x>0,'Attenzione: x negativo (x={0})'.format(x)
```

```
---------------------------------------------------------------------------
AssertionError                            Traceback (most recent call last)
<ipython-input-7-58ed7a6a7cb0> in <module>()
      5 x = -1
      6
----> 7 assert x>0,'Attenzione: x negativo (x={0})'.format(x)

AssertionError: Attenzione: x negativo (x=-1)
```

Ciò equivale a:

In [8]:
```
if x<0:
    raise AssertionError('Attenzione: x negativo (x={0})'.format(x))
```

```
---------------------------------------------------------------------------
AssertionError                            Traceback (most recent call last)
<ipython-input-8-787dbc8f3428> in <module>()
      1 if x<0:
----> 2     raise AssertionError('Attenzione: x negativo (x={0})'.format(x))

AssertionError: Attenzione: x negativo (x=-1)
```

salvo che per il fatto che l'eccezione viene sollevata anche nel caso di interprete funzionante in modalità ottimizzata.

NOTA: l'effetto di assert è quello di sollevare un eccezione e quindi non ha senso usare questo costrutto quando un'eccezione verrebbe comunque ed immediatamente sollevata , come nell'esempio sotto:

In [ ]:
```
def indicizza(l,i):
    assert i>= len(l),'Fuori margini di l' ## inutile!!
    return l[i]
```