Lezione 10 – Alberi e gestione delle eccezioni

Informatica

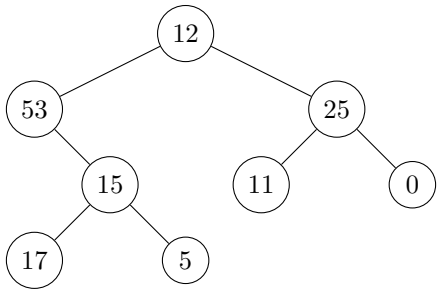
5 Maggio 2016

1 Visita in ampiezza ed esercizi

Esercizio: stampa per livelli

Scrivere una funzione che stampa le etichette di tutti i nodi dell'albero *per livelli*: prima la radice, poi tutti i nodi a livello 1, quelli a livello 2, eccetera ...

Esempio: dato l'albero binario



la funzione deve stampare `12 53 25 15 11 0 17 5`

Una possibile soluzione modulare divide il problema in due parti:

- crea una lista contenente tutti i nodi al livello `n`
- stampa un livello dopo l'altro

```
def livello(t, n):
    if is_empty(t):
        return []
    elif n == 0:
        return [label(t)]
    else:
        return livello(first_child(t), n-1) + livello(second_child(t), n-1)

def stampa_livelli(t):
    n = 0
    l = livello(t, n)
    while l != []:
        for e in l:
            print(e, end=" ")
        n = n + 1
        l = livello(t, n)
    print()

print("---- LIVELLI ----")
```

```
stampa_livelli(t)
stampa_livelli(tree())
stampa_livelli(leaf(1))
```

Tuttavia, questa soluzione non è efficiente: l'albero viene visitato più volte. Per essere precisi, effettua un numero di visite dell'albero pari al numero dei livelli.

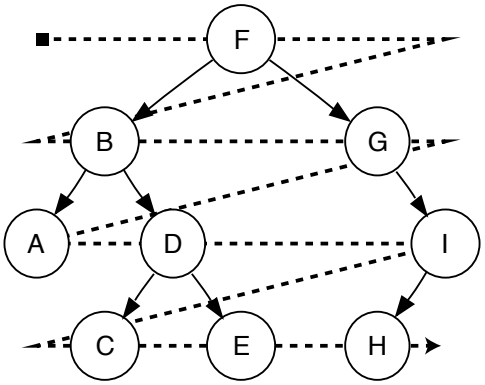
Esiste un modo per risolvere il problema visitando l'albero una sola volta?

La visita in ampiezza

Abbiamo un problema . . .
. . . nessuna delle tre visite che abbiamo visto finora (infissa, prefissa e postfissa) è in grado di risolvere l'esercizio!

. . . dobbiamo usare un *nuovo tipo di visita* . . .

Visita in ampiezza
Visita ciascun livello dell'albero da sinistra verso destra.



Per effettuare una visita in ampiezza dobbiamo usare una *coda* che ci permetterà di mantenere l'ordine dei livelli. La coda conterrà i sottoalberi che dobbiamo ancora visitare. Il procedimento è sintetizzato come segue:

1. Mettere in coda l'albero iniziale.
2. Togliere dalla coda un albero (nella prima iterazione l'albero iniziale) ed esaminarlo.
3. Se l'albero corrente è vuoto, non fare nulla
4. Se non è vuoto, visitare la radice e mettere in coda il sottoalbero sinistro seguito dal sottoalbero destro
5. Se la coda non è vuota, ripetere il passo 2.

```
import coda

def breadth_visit(t):
    q = coda.queue()
    coda.enqueue(t, q)
    while not coda.is_empty(q):
        s = coda.dequeue(q)
        if not is_empty(s):
            print(label(s), end=" ")
            coda.enqueue(first_child(s), q)
            coda.enqueue(second_child(s), q)
```

```
    print()

print("---- AMPIEZZA ----")
t = bin(12,
        bin(53, tree(), bin(15, leaf(17), leaf(5))),
        bin(25, leaf(11), leaf(0)))
breadth_visit(t)
breadth_visit(tree())
breadth_visit(leaf(1))
```

Esercizi sugli alberi

1. Un albero binario *pieno* è un albero binario in cui tutti i nodi hanno esattamente 0 o 2 figli, e nessun nodo ha 1 figlio. Scrivere una funzione `pieno(t)` che ritorna `True` sse l'albero `t` è pieno.
2. Scrivere una funzione `inverti(t)` che dato un albero binario `t` restituisce l'albero inverso, ovvero un albero in cui *per ogni nodo* il figlio destro (con relativo sottoalbero) è scambiato con il figlio sinistro (con relativo sottoalbero).
3. Dato un albero binario i cui nodi contengono interi, si vuole aggiungere ad ogni foglia un figlio contenente la somma dei valori che appaiono nel cammino dalla radice a tale foglia. Scrivere una funzione `aggiungi_somma(t)` che esegua questa operazione.

```
def pieno(t):
    if is_empty(t):
        return True
    elif is_leaf(t):
        return True
    else:
        fc = first_child(t)
        sc = second_child(t)
        if is_empty(fc):
            return False
        if is_empty(sc):
            return False
        return pieno(fc) and pieno(sc)

print("---- PIENO ----")
print(pieno(t))
print(pieno(tree()))
print(pieno(leaf(1)))
print(pieno(bin(1, leaf(1), bin(2, leaf(3), leaf(4)))))

def inverti(t):
    if is_empty(t):
        return tree()
    fc = first_child(t)
    sc = second_child(t)
    r = label(t)
    return bin(r, inverti(sc), inverti(fc))

print("---- INVERTI ----")
print(inverti(t))
print(inverti(tree()))
print(inverti(leaf(1)))
```

```
print(inverti(bin(1, leaf(1), bin(2, leaf(3), leaf(4)))))

def aggiungi_somma(t):
    return aggiungi_somma_aux(t, 0)

def aggiungi_somma_aux(t, n):
    if is_empty(t):
        return t
    elif is_leaf(t):
        r = label(t)
        return bin(r, leaf(n+r), tree())
    else:
        r = label(t)
        return bin(r,
                   aggiungi_somma_aux(first_child(t), n+r),
                   aggiungi_somma_aux(second_child(t), n+r))

print("---- AGGIUNGI SOMMA ----")
print(aggiungi_somma(t))
print(aggiungi_somma(tree()))
print(aggiungi_somma(leaf(1)))
print(aggiungi_somma(bin(1, leaf(1), bin(2, leaf(3), leaf(4)))))
```

2 Gestione delle eccezioni in Python

Un programma semplicissimo

Consideriamo un semplice programma che fa la divisione tra due numeri inseriti in input dall'utente:

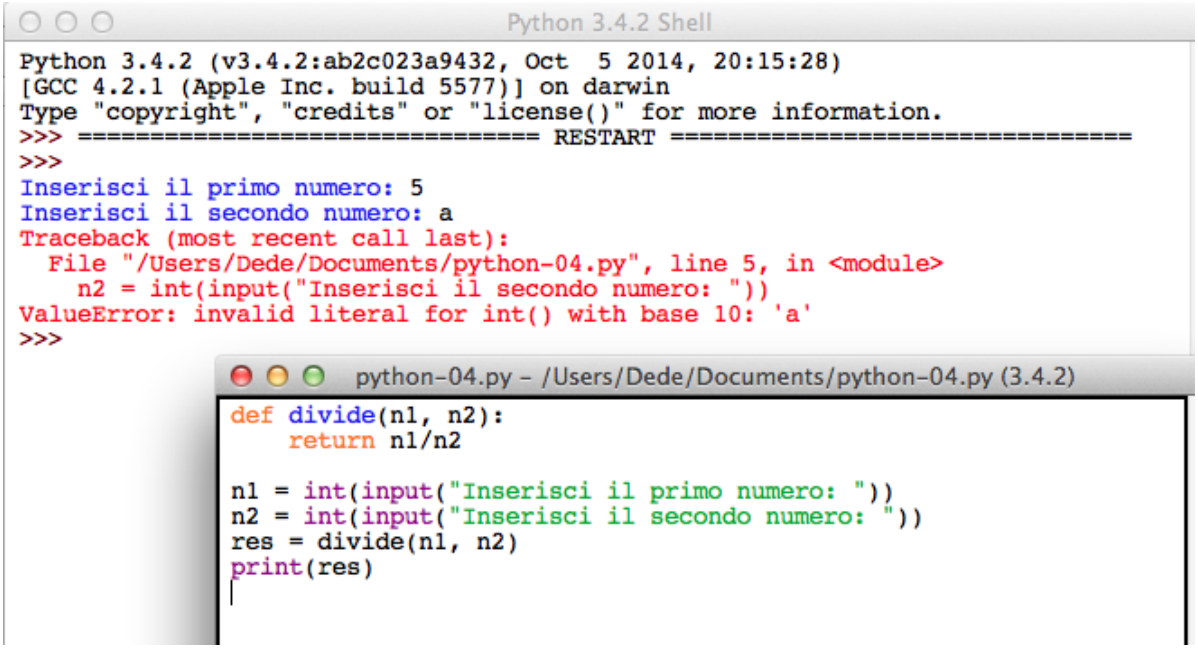
```
def divide(n1, n2):
    return n1/n2

n1 = int(input("Inserisci il primo numero: "))
n2 = int(input("Inserisci il secondo numero: "))
res = divide(n1, n2)
print(res)
```

Cosa succede quando l'utente inserisce in input *un valore che non rappresenta un numero intero*, come ad esempio un carattere?

Da qui in poi, esempi di codice e figure by *Davis Molinari, www.davismol.net*

Si genera un errore!



Esiste un modo per evitare l'errore?

Il costrutto try/except

Python mette a disposizione un meccanismo per la *gestione delle eccezioni* costituito dal costrutto try/except:

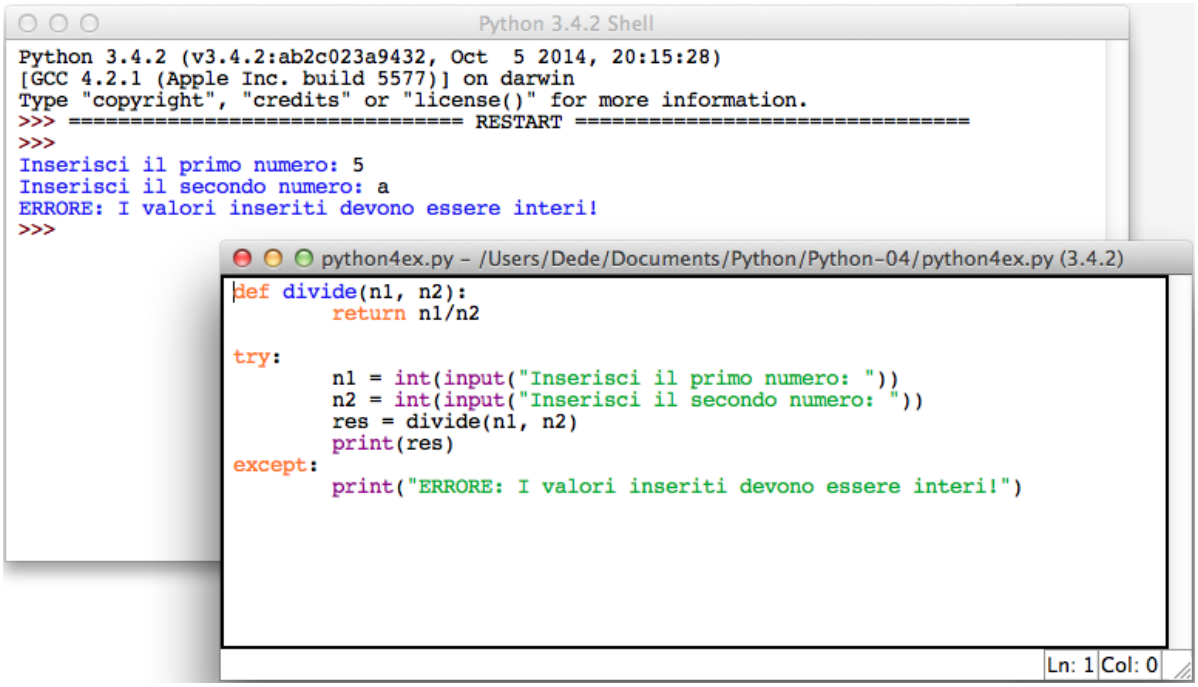
```
def divide(n1, n2):
    return n1/n2

try:
    n1 = int(input("Inserisci il primo numero: "))
    n2 = int(input("Inserisci il secondo numero: "))
    res = divide(n1, n2)
    print(res)
except:
    print("ERRORE: I valori inseriti devono essere interi!")
```

Se l'utente inserisce un valore non valido, non si ottiene l'errore:

- viene eseguito *il codice all'interno del blocco* `except` che mostra un messaggio personalizzato per segnalare all'utente che i valori che ha inserito non sono validi.

Proviamo!



L'inserimento di un valore non numerico però non è l'unico problema che si può presentare. Se ad esempio l'utente inserisce come secondo valore uno 0, la divisione produce nuovamente un errore e non possiamo però mostrare lo stesso messaggio che indica di inserire un intero ma dobbiamo mostrare un messaggio di errore differente.

`except` e tipi di errore

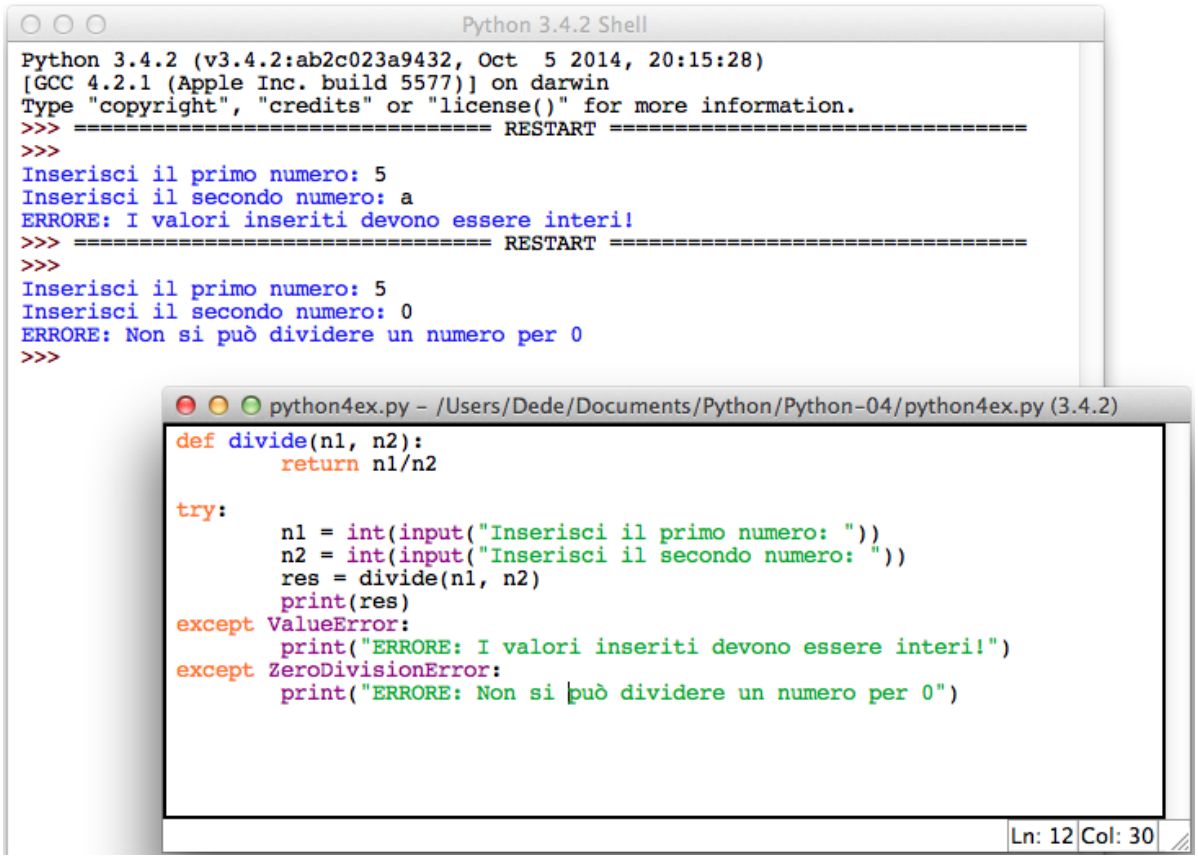
Per gestire tipi diversi di errore che possono essere scatenati dallo stesso blocco di codice occorre specificare nello statement except il *tipo di eccezione* che si vuole intercettare

```
def divide(n1, n2):
    return n1/n2

try:
    n1 = int(input("Inserisci il primo numero: "))
    n2 = int(input("Inserisci il secondo numero: "))
    res = divide(n1, n2)
    print(res)
except ValueError:
    print("ERRORE: I valori inseriti devono essere interi!")
except ZeroDivisionError:
    print("ERRORE: Non si può dividere un numero per 0")
```

In generale `except` senza l'indicazione del tipo di errore non deve mai essere usato!

Proviamo!



Tipi di eccezioni

`ZeroDivisionError`	divisione per zero
`IndexError`	indice di una sequenza errato
`KeyError`	chiave non presente nel dizionario
`TypeError`	operazione o funzione applicata ad un tipo non appropriato
`ValueError`	operazione o funzione applicata ad un oggetto di tipo corretto ma valore inappropriato
`NameError`	nome locale o globale non trovato
`OSError`	errore causato dal sistema operativo (p.es. nella lettura/scrittura di un file)

L'elenco completo dei tipi di eccezione si trova *QUI*

try/except/else

Il costrutto try/except prevede una *clausola opzionale* `else` che permette di definire il blocco di codice da eseguire *quando l'eccezione non si verifica*

```
def divide(n1, n2):
    return n1/n2

try:
    n1 = int(input("Inserisci il primo numero: "))
    n2 = int(input("Inserisci il secondo numero: "))
except ValueError:
    print("ERRORE: I valori inseriti devono essere interi!")
else:
    try:
        res = divide(n1, n2)
    except ZeroDivisionError:
        print("ERRORE: Non si può dividere un numero per 0")
    else:
        print(res)
```

Proviamo!