Esempi di esercizi per la prova pratica sul linguaggio Python

December 20, 2019

Gli esercizi inclusi in queste note sono "puramente" indicativi delle prove di programmazione Python che lo studente dovrà affrontare in sede di esame. Non è lecito attendersi che queste ultime possano essere esclusivamente semplici modifiche degli esercizi qui presentati. Ciò che è invece corretto attendersi è che le suddette prove d'esame possano essere svolte col solo bagaglio di conoscenze acquisite durante le lezioni e le esercitazioni in classe.

All'inizio di ogni esercizio è presente un numero racchiuso fra parentesi quadre. Si tratta di un'indicazione, su scala [1-10], della difficoltà che il docente attribuisce all'esercizio in questione. Dovrebbe essere superfluo sottolineare che tale misura è solo una stima soggettiva (del docente) che potrebbe non coincidere con la difficoltà percepita dallo studente in sede d'esame, quest'ultima essendo chiaramente funzione della preparazione.

A quest'ultimo riguardo vanno comunque fatte due precisazioni:

1. La stima della difficoltà riguarda una soluzione "minima funzionante";
2. efficienza, eleganza e "pythonicità" della soluzione saranno elementi di valutazione degli elaborati ma **in nessun caso** saranno elementi discriminanti per il superamento dell'esame. In parole più chiare, alla sufficienza si può arrivare anche prescindendo da questi aspetti.

Fatte tutte queste premesse, si precisa che la prova d'esame sarà basata su un esercizio di difficoltà stimata fra 1 e 5 e un'esercizio nel range 6-10.

Infine, per alcuni esercizi sono qui presentate possibili soluzioni, come aiuto alla preparazione dell'esame. Si tenga tuttavia presente che, in alcuni casi, le soluzioni "vanno oltre" rispetto a quanto sopra ricordato riguardo il minimo funzionante.

1 Python basics

1. [1] Scrivere una funzione che riceve come parametri due liste di numeri interi e restituisce una terza lista contenente gli interi dispari della prima e gli interi pari della seconda. In caso di errore di tipo nelle liste di ingresso la funzione deve restituire `False`.

   Una soluzione

```
def combinelist(L1,L2):
    if not (isinstance(sum(L1),int) and \
            isinstance(sum(L2),int)):
        return False
    return [x for x in L1 if x%2==1] + \
           [x for x in L2 if x%2==0]
```

2. [1] Scrivere una funzione che riceve in ingresso due dizionari `A` e `B` e restituisce un dizionario `C` così formato: (1) le chiavi di `C` sono (tutte e sole) le chiavi presenti sia in `A` che in `B`; (2) se k è una chiave di `C`, allora `C`$[k] = (v_{Ak}, v_{Bk})$, dove v_{Ak} e v_{Bk} sono rispettivamente di valori corrispondenti a k in `A` e in `B`.

3. [3] Scrivere una funzione che riceve in ingresso due insiemi `A` e `B` di numeri interi positivi e, opzionalmente, un intero `L`; la funzione rimuove da `A` tutti gli elementi maggiori di `L` che appartengono anche a `B`. Se `L` non è specificato, il filtro non si applica. In caso di errore di tipo sugli elementi da rimuovere (cioè se non sono interi), la funzione deve stampare un messaggio opportuno e lasciare `A` inalterato.

   Una soluzione

```
def filter(A,B,L=0):
    try:
        S = [(x<<1)>>1 for x in B&A if x>L]
    except TypeError:
        print("""Errore di tipo:
l'insieme non viene modificato""")
        return
    for s in S:
        A.remove(s)
```

4. [5] Fornire almeno tre implementazioni "sostanzialmente" differenti per la funzione che, ricevuta in input una lista L, restituisce l'insieme degli elementi di L che sono istanza di `int`.

   Una soluzione

```
def filter2(A):
    return set(map(lambda a: a if isinstance(a,int) \
              else -1,A)).union({-1}).difference({-1})

def filter3(A):
    X = set()
    for a in A:
        if isinstance(a,int):
            X.add(a)
    return X

def filter4(A):
    if A == []:
        return set()
    a = A[0]
    if isinstance(a,int):
        return filter4(A[1:]).union({a})
    else:
        return filter4(A[1:])
```

2 Attributes, Properties, and descriptors

5. [8] Scrivere una classe che preveda un contatore per gli accessi ad ogni attributo. Il contatore deve essere incrementato sia per gli accessi in lettura sia per quelli in scrittura. La classe deve essere istanziata scrivendo gli attributi come parametri keyword.

Una soluzione

```
class CountAttr:
    ''' Usa un dizionario per memorizzare il numero di
    accessi agli attributi. Questo significa che esiste
    nel dizionario una chiave 'x' per ogni attributo x.
    Per poter registrare ogni accesso vengono riscritti
    i metodi __getattribute__ e __setattr__ , che possono
    facilmente portare al problema delle chiamate
    ricorsive illimitate. Per questo, all'interno della
    classe, si deve accedere agli attributi utilizzando
    i corrispondenti metodi in object. Attenzione: se
    eseguito sotto ipython questo "programma" evidenzia
    accessi al dizionario della classe che non dipendono
    dal codice. '''

    def __init__(self,**kwargs):
        object.__setattr__(self,'_count',{})
        for k,v in kwargs.items():
            object.__setattr__(self,k,v)
            object.__getattribute__(self,'_count')[k] = 0

    def __getattribute__(self,attr):
        d = object.__getattribute__(self,'_count')
        x = d.get(attr,None)
        if x is None:
            print(f"L'attributo {attr} non e' definito")
            return None
        x += 1
        d[attr] = x
        return object.__getattribute__(self,attr)

    def __setattr__(self,name,value):
        x = object.__getattribute__(self,'_count').get(
                                        name,None)
        x = 0 if x is None else x+1
        object.__getattribute__(self,'_count')[name] = x
        object.__setattr__(self,name,value)
```

6. [7] Si definisca una classe `convertitore` che possa essere utilizzata come descrittore e che effettui conversioni di valuta. Si supponga al riguardo di avere un dizionario del tipo:

```
cambio = {'euro-dollaroUSA': 1.11,
          'dollaroUSA-euro': 0.90,
          'euro-sterlinaGB': 0.85,
          'sterlinaGB-euro': 1.18,
          'dollaroUSA-sterlinaGB': 0.76,
          'sterlinaGB-dollaroUSA': 1.31}
```

e una classe `valori`, che rappresenta una data quantità di denaro in una data valuta, così definita:

```
class valori:

    def __init__(self,valuta,importo):
        self.valuta = valuta
        self.importo = importo

    altrevalute = convertitore(cambio)

    def controvalore(self):
        for v,c in self.altrevalute:
            print(f"{v:.2f}\t{c}")
```

Il descrittore deve essere realizzato in modo che il valore dell'attributo `altrevalute` sia una lista di coppie: `(valuta,controvalore)`, dove il controvalore è il corrispondente nella `valuta` del valore di `self.valuta`. Ad esempio, se `self.valuta` fosse `Euro` e `self.importo` fosse 10, il valore di `altrevalute` dovrebbe essere una lista con il corrispondente di 10 euro in dollari e sterline.

Una soluzione

```
class convertitore:
    def __init__(self,cambio):
        self.cambio = cambio

    def __get__(self,inst,cls):
        c = inst.valuta
        v = inst.importo
        keys = [k for k in self.cambio.keys() \
                if k.find(f"{c}-")!=-1]
        return [(v*self.cambio[k],k.split('-')[1]) \
                  for k in keys]
```

3 Classes

7. [5] Implementare un'opportuna sottoclasse di `list` costituita solo da liste di interi e che supporti il confronto così definito: $L_1 == L_2$ se e solo se la somma degli elementi di L_1 è uguale della somma degli elementi di L_2. Sulla base di questo criterio implementare anche tutti gli altri operatori relazionali.

   Una soluzione

```
class weirdlist(list):
    def __new__(cls,iterable):
        L = list(iterable)
        try:
            isinstance(sum(L),int)
            return super().__new__(cls,L)
        except TypeError:
            print("Non tutti gli elementi sono interi")
            return None

    def __eq__(self,other):
        return sum(self)==sum(other)

    def __ne__(self,other):
        return sum(self)!=sum(other)

    def __lt__(self,other):
        return sum(self)<sum(other)

    def __leq__(self,other):
        return sum(self)<=sum(other)

    def __gt__(self,other):
        return sum(self)>sum(other)

    def __geq__(self,other):
        return sum(self)>=sum(other)
```

8. [6] Definire una classe per operare sui numeri complessi. Un numero complesso deve essere visto e trattato come coppia di numeri reali rappresentanti, rispettivamente, parte reale e parte immaginaria. La classe deve supportare (come minimo) le operazioni di somma e prodotto di numeri complesso, oltre al calcolo del coniugato di un numero complesso.

4 Iterators and generators

9. [4] I cosiddetti numeri *Catalani* sono così definiti:

$$C_n = \frac{1}{n+1}\binom{2n}{n} \qquad n = 1, 2, \ldots$$

Implementare un iterabile per i primi n numeri Catalani.

Una soluzione (NB: La difficoltà attribuita a questo esercizio è, come sempre, relativa alla soluzione "minima funzionante", che può ben essere rappresentata da questa prima classe)

```
class Catalan:
    '''Iteratore per i primi n numeri Catalani.
       Versione inefficiente per via del ricalcolo
       completo del j-esimo numero catalano, j = 0,...,n
    '''
    def __init__(self,n):
        assert n>=0
        self.n = n
        self.j = 0

    def __iter__(self):
        return self

    def __next__(self):
        from math import factorial
        if self.j > self.n:
            raise StopIteration
        f = factorial(self.j)
        C = int(1/(self.j+1)*factorial(2*self.j)/(f*f))
        self.j += 1
        return C
```

Una soluzione più efficiente

```
class Catalan:
    '''Ogni numero viene calcolato a partire dal
    precedente con due operazioni moltiplicative'''
    def __init__(self,n):
        assert n>=0
        self.n = n
        self.j = 0
        self.C = 1
        self.stop = False

    def __iter__(self):
        return self

    def __next__(self):
        if self.j>self.n:
            raise(StopIteration)
        C = self.C
        self.C = int(C*((self.j<<2)+2)/(self.j+2))
        self.j+=1
        return C
```

10. [9] Implementare un generatore per i primi n numeri primi (1 escluso).

Una soluzione

```
def genprime(n):
    from math import sqrt
    first_primes = [2,3,5,7,11,13,17,19,23,29]
    for i in range(n):
        if i < len(first_primes):
            yield first_primes[i]
        else:
            gotit = False
            p = first_primes[-1]
            while not gotit:
                p+=2; i = 1
                M = int(sqrt(p))
                while first_primes[i]<=M and \
                      p%first_primes[i]!=0:
                    i+=1
                if first_primes[i]>M:
                    first_primes.append(p)
                    gotit = True
            yield p
```

5 Function decorators

11. [1] Scrivere un semplice decoratore `greetings` per stampare, all'inizio di ogni chiamata di funzione, il messaggio: `Hi, this is` ⟨nome della funzione⟩.

12. [3] In relazione all'esercizio 3, definire un decoratore in cui sono inseriti i controlli di tipo (togliendoli qundi dalla funzione `filter`).

13. [4] Scrivere una classe per decorare funzioni con l'aggiunta di una proprietà che memorizza il numero di volte che la funzione è stata chiamata.

    Una soluzione (NB La stampa non è a rigori richiesta)

```
class count:
    def __init__(self,fun):
        self.n = 0
        self.fun = fun
        self.name = fun.__name__

    def __call__(self,*args,**kwargs):
        self.n += 1
        return self.fun(*args,**kwargs)

    def numcalls(self):
        print(f"Function {self.name}",end=' ')
        print(f"has been called {self.n} times")
```

14. [3] Data la funzione:

```
def div(a,b):
  return int(a/b),a%b
```

    che, 'idealmente", calcola quoziente e resto di una divisione intera, definire un decoratore che implementa i controlli necessari affinché il risultato sia corretto o venga prodotto un adeguato messaggio di errore.

Una soluzione

```
def controllo(f):
    def intdiv(a,b):
        if not isinstance(a,int) or \
            not isinstance(b,int):
            print("Operandi non interi")
            return None
        if b == 0:
            print("Divisione per zero")
            return None
        return f(a,b)
    return intdiv

@controllo
def div(a,b):
    return int(a/b),a\%b
```

6 Metaprogramming

15. [5] Scrivere una metaclasse che possa essere il tipo di classi che "contano" il numero di oggetti creati.

    Una soluzione

```
class mymeta(type):

    def __init__(self,*args,**kwargs):
        self.count_objects = 0

    def __call__(cls,*args,**kwargs):
        cls.count_objects += 1
        return cls.__new__(cls,*args,**kwargs)

class A(metaclass=mymeta):
    pass
```

16. [2] Si consideri un file in cui, in ogni riga, è presente una sequenza di numeri separati da virgole (più eventuali spazi. Si scriva una funzione che legge le righe del file e stampa tutte le somme (cioè la somma dei numeri in ogni riga).

    Una soluzione (elegante ma pericolosa)

```
def sumrows(fn):
    with open(fn,'r') as f:
        for line in f:
            print(eval(line.replace(',','+')))
```