python_2

October 3, 2018

1 Variabili, namespace e regole di scopo

1.1 Variabili e namespace

Una variabile in Python è il nome per un oggetto (funzionalmente possiamo dire che è una associazione fra nomi e oggetti). Le variabili Python sono quindi diverse dalle variabili C o Java.

1. Non hanno tipo ma acquisiscono il tipo dell'oggetto denotato
2. Un nome è dichiarato come variabile nel momento in cui viene usato (a sinistra) in un assegnamento
3. In ogni particolare punto del programma un nome può corrispondere a una variabile:
    1. locale
    2. non locale
    3. globale
    4. predefinita (***built-in***)
    5. ... e naturalmente può anche non corrispondere ad alcuna variabile definita
4. In ogni particolare punto del programma i nomi locali, globali e predefiniti formano altrettante "collezioni" chiamate ***namespace***
5. In Python i namespace sono essi stessi oggetti manipolabili (sono ***dizionari***) ...
6. ... e come tali possono essere consultati

```
In [ ]: print(dir()) # Restituisce i nomi (chiavi) nel namespace corrente
```

```
In [ ]: print(locals()['__name__'])
```

```
In [ ]: __name__
```

```
In [ ]: # A livello di prompt dell'interprete il namespace corrente è quello globale
        if sorted(dir())==sorted(globals().keys()):
            print("Il namespace corrente è quello globale")
```

```
In [ ]: def unaltrafunzione(x):
            x = 10
            print("Il valore locale di x è {}".format(locals()['x']))
            print("Il valore globale di x è {}".format(globals()['x']))
        x = 1
        unaltrafunzione(x)
```

```
In [ ]: # All'interno di una funzione il namespace corrente è quello locale alla funzione stes
        def unafunzione(x):
            a = 5
            b = -3
            x = a*x+b
            if sorted(dir())==sorted(locals().keys()):
                print("Il namespace corrente è quello locale alla funzione")
            return x

        x = 1
        print(unafunzione(x))
        print(x)
```

1.2 Regole di *scopo* (*scope rules*)

1. Lo ***scope*** di una (dichiarazione di) variabile sono i punti del programma in cui tale variabile può essere acceduta
2. L'interprete usa la ***regola LEGB*** per determinare qual è la variabile (e dunque il valore) cui si riferisce un dato nome:
    1. Prima il namespace ***locale*** (***L***)
    2. Poi i namespace locali alle ***enclosing*** function (***E***)
    3. Poi il namespace ***globale*** (***G***)
    4. Infine il namespace delle funzioni ***built-in*** (***B***)

```
In [ ]: # Un po' di prove....
        def esterna(x):
            a = 5           # Introduce nome locale
            b = x           # Introduce nome locale
            def interna():
                global a
                nonlocal b
                x = a+b     # a(globale)+b(nonlocale)
                a = 2       # Altera il nome globale a
                b = 5       # Altera il nome non locale b
                return x
            a = interna()   # Altera il nome locale a
            print(a)
            print(b)
            return a+b
        a = 1
        print(a,esterna(a),a)
```

1.3 Come agiscono le importazioni di moduli sui namespace?

1.3.1 La sintassi: from *modulo* import *nome*

1.3.2 include *nome* nel namespace corrente

```
In [ ]: print(globals().get('log','non definita')) # Pythonic
        from math import log
        print(globals().get('log','non definita'))
```

1.3.3 Ogni modulo ha però il proprio namespace

```
In [ ]: print(globals().get('log2','non definita'))
        import math
        print(globals().get('log2','non definita'))
        if 'log2' in dir(math):
            print(math.log2)
        else:
            print('non definita')
```

1.3.4 Quindi attenzione:

```
In [ ]: import math
        def fail(x):
            return log10(x)
        print(fail(5))
```

```
In [ ]: import math
        def success(x):
            return math.log10(x)
        print(success(5))
```

1.4 Oggetti, attributi e proprietà

1.4.1 Gli oggetti Python (*tutti* gli oggetti Python) possono avere *attributi*. Gli attributi, come le variabili, sono a loro volta associazioni fra nomi e oggetti.

1.4.2 Gli attributi (insieme ai *metodi*) caratterizzano un oggetto

1.4.3 La funzione dir() si può applicare anche ad un oggetto e restituisce una lista che elenca, sotto forma di stringhe, tutti i suoi attributi

```
In [ ]: # Un "semplice" numero intero ha moltissimi attributi
        print(len(dir(3)))
        # Ne elenchiamo il sedicesimo ...
        print(dir(3)[15])
        # ... i primi 3 ...
        print(dir(3)[:3])
        # ... e gli ultimi 5
        print(dir(3)[-5:])
```

```
        # Sono attributi ereditati
        dir(3)==dir(int)
```

1.4.4 *getattr* e *setattr*

```
In [ ]: L=[1,2,3]
        print(dir(L)[:7])
```

```
In [ ]: ### Due "modi" diversi per accedere agli attributi
        print(L.__add__)
        print(getattr(L,'__add__'))
```

```
In [ ]: L.__add__([4,5])
```

```
In [ ]: getattr(L,'__add__')([6,7,8])
```

```
In [ ]: L
```

1.4.5 Gli attributi degli oggetti dei tipi predefiniti in Python sono read-only e non si possono aggiungere attributi nuovi

```
In [ ]: L=[1,2,3]
        def myadd(L,M):
            T = M[:]
            T.reverse()
            for t in T:
                L.append(t)         # Voglio che sia modificata L
            return L
        setattr(L,'__add__',myadd)
```

1.4.6 Si possono però (ovviamente...) modificare/aggiungere attributi a tipi definiti da utente come pure a classi derivate

```
In [ ]: # Pythonic
        class MyList(list):
            pass

        setattr(MyList,'__add__',myadd)
```

```
In [ ]: L = MyList([1,2])
        M = [4,3]
        L.__add__(M)
```

```
In [ ]: L+M              /* Zucchero sintattico; viene chiamato il metodo __add__ di L
```

1.5 Python consente di definire il significato degli operatori per tipi definiti da utente

2 Un primo progetto completo: MST su grafo pesato

```
In [ ]: class edge:
            '''Classe che rappresenta archi non orientati pesati, v1.0 (non pythonica...).
```

```
        L'arco vero e proprio, cioè la coppia (u,v) di vertici,
        e il suo peso sono messi 'sullo stesso piano', sono cioè proprietà
        dell'oggetto arco.
        '''
    def __init__(self,e,w):
        '''Il parametro e è una coppia (tupla Python): e=(u,v)
           mentre w è il peso.'''
        self.e = e
        self.w = w

    def lt(self,other):
        '''Metodo che confronta l'arco self con other, sulla base
           prima del peso e poi dell'ordine lessicografico'''
        ws = self.w
        wo = other.w
        ms = min(self.e)
        mo = min(other.e)
        if ws<wo or (ws==wo and ms<mo) or \
            (ws==wo and ms==mo and max(self.e)<max(other.e)):
            return True
        else:
            return False
```

In [ ]:
```
# Esempio
e1 = edge((2,1),3)
e2 = edge((1,5),3)
e1.lt(e2)
```

In [ ]:
```
# Usiamo il semplice insertion sort
def edgesort(E):
    '''Usa insertion sort per ordinare una lista di archi'''
    n = len(E)
    for i in range(1,n):
        e = E[i]
        j = i-1
        while j>=0 and e.lt(E[j]):
            E[j+1] = E[j]
            j = j-1
        E[j+1] = e
```

In [ ]:
```
E = [edge((2,1),3),edge((1,5),3),edge((2,3),1),edge((2,4),1)]
```

In [ ]:
```
# Come facciamo a "vedere" se l'algoritmo funziona correttamente?
print(E)
# come pure
for e in E:
    print(e)
```

2.1 Ancora un *magic method* e ancora zucchero sintattico

```
In [ ]: if 'edge' in dir():
            del(edge)

        class edge:
            '''Classe che rappresenta archi non orientati pesati, v1.0 (non pythonica...).
               L'arco vero e proprio, cioè la coppia (u,v) di vertici,
               e il suo peso sono messi 'sullo stesso piano', sono cioè proprietà
               dell'oggetto arco.
               '''
            def __init__(self,e,w):
                '''Il parametro e ''è una coppia (tupla Python): e=(u,v)
                   mentre w è il peso.'''
                self.e = e
                self.w = w

            def __str__(self):
                '''Definisce la rappresentazione di oggetti edge. Usata da print() e str()'''
                s = str(self.e)+": "+str(self.w)
                return s

            def lt(self,other):
                '''Metodo che confronta l'arco self con other, sulla base
                   prima del peso e poi dell'ordine lessicografico'''
                ws = self.w
                wo = other.w
                ms = min(self.e)
                mo = min(other.e)
                if ws<wo or (ws==wo and ms<mo) or \
                    (ws==wo and ms==mo and max(self.e)<max(other.e)):
                    return True
                else:
                    return False
```

```
In [ ]: # Abbiamo ora una rappresentazione "comprensibile" degli edge
        e=edge((1,2),9)
        print(e)
```

```
In [ ]: # Attenzione però:
        E = [edge((2,1),3),edge((1,5),3),edge((2,3),1),edge((2,4),1)]
        print(E)
```

2.2 Possiamo ora vedere se l'ordinamento funziona (almeno su un esempio)

```
In [ ]: print("Prima:")
        for e in E:
            print(e)
        edgesort(E)
```

```
print("Dopo")
for e in E:
    print(e)
```

2.3 I magic method permettono anche di (ri)definire il comportamento degli operatori relazionali. Ad esempio, il comportamento di == è controllabile mediante il metodo _eq_, mentre il comportamento di < è controllabile da _leq_.

2.4 Sarebbe quindi "bello" che il confronto fra archi potesse essere espresso come

2.5 *e < E[j]*

2.6 anziché come

2.7 *e.leq(E[j])*

```
In [ ]: if 'edge' in dir():
            del(edge)

        class edge:
            '''Classe che rappresenta archi non orientati pesati, v1.11 (non pythonica...).
               L'arco vero e proprio, cioè la coppia (u,v) di vertici,
               e il suo peso sono messi 'sullo stesso piano', sono cioè proprietà
               dell'oggetto arco.
               '''
            def __init__(self,e,w):
                '''Il parametro e ''è una coppia (tupla Python): e=(u,v)
                   mentre w è il peso.'''
                self.e = e
                self.w = w

            def __str__(self):
                '''Definisce la rappresentazione di oggetti edge. Usata da print() e str()'''
                s = str(self.e)+": "+str(self.w)
                return s

            def __lt__(self,other):
                '''Definire __lt__ forza un nuovo comportamento di <.
                   La funzione è identica al precedente metodo lt, con
                   la sola eccezione della stampa.'''
                print("Chiamato il metodo magico __lt__") # Se vediamo questa stampa... il met
                ws = self.w
                wo = other.w
                ms = min(self.e)
                mo = min(other.e)
                if ws<wo or (ws==wo and ms<mo) or \
                    (ws==wo and ms==mo and max(self.e)<max(other.e)):
                    return True
                else:
```

```
                    return False
```

```
In [ ]: # Come metodo funziona
        e1 = edge((5,1),3)
        e2 = edge((1,2),1)
        e1.__lt__(e2)
```

```
In [ ]: # E addirittura...
        e1<e2
```

2.8 L'ultima soluzione è più pythonica...

2.9 Nella soluzione attuale gli "oggetti" edge hanno due attributi, di cui uno (guarda caso) è l'edge vero e proprio mentre l'altro è il peso dell'edge.

2.10 La soluzione pythonica prevede di avere un oggetto edge con proprietà peso. In altri termini, un edge sarà la coppia (u,v) mentre il peso w sarà una sua proprietà. La classe edge viene dunque definita come sottoclasse di *tuple*.

```
In [ ]: if 'edge' in dir():
            del(edge)

        class edge(tuple):
            '''Classe che rappresenta archi non orientati pesati, v2.0.
            Eredita da tuple.
               '''
            def __new__(cls,e,w):
                '''Ridefiniamo il metodo __new__ perché abbiamo bisogno di specificare
                   un parametro addizionale e non solo la coppia e=(u,v)'''
                edge = super().__new__(cls,e)
                edge.w = w
                return edge

            def __str__(self):
                '''Definisce la rappresentazione di oggetti edge. Usata da print() e str().
                   Occhio alle differenze con le versioni precedenti.'''
                s = super().__str__()+": "+str(self.w)
                # s = tuple.__str__(self)     # Alternativa, meno in linea con Python3
                return s

            def __lt__(self,other):
                '''Definire __lt__ forza un nuovo comportamento di <.
                   Metodo che confronta l'arco self con other, sulla base
                   prima del peso e poi dell'ordine lessicografico.
                   Anche qui, occhio alle differenze con le versioni precedenti'''
                print("Chiamato il metodo magico __lt__")
                ws = self.w
                wo = other.w
                ms = min(self)
```

```
        mo = min(other)
        if ws<wo or (ws==wo and ms<mo) or \
            (ws==wo and ms==mo and max(self)<max(other)):
            return True
        else:
            return False
```

```
In [ ]: # Il metodo magico viene chiamato quando viene usato l'operatore < per confrontare edg
        e1 = edge((5,1),3)
        e2 = edge((1,2),1)
        e1 < e2
```

2.11 **Possiamo ora procedere con il progetto. Ci serve una funzione (ma una classe sarebbe meglio...) per leggere il grafo da file**

2.12 **Definiamo innanzi tutto una semplice rappresentazione esterna**

2.13 **Nell'ipotesi che i vertici siano individuati da (o comunque che siano in corrispondenza con) i numeri 1, 2, ..., n, possiamo rappresentare un grafo (non orientato) pesato come lista di terne, una terna per riga:**

$u_1 \quad v_1 \quad w_1$
$u_2 \quad v_2 \quad w_2$
$\ldots$
$u_m \quad v_m \quad w_m$

2.14 **dove gli u_i e i v_i sono numeri interi nell'intervallo $[1, n]$, mentre i w_i sono arbitrari numeri reali (non negativi, nella maggior parte delle applicazioni) che rappresentano i pesi.**

2.15 **Il numero di archi del grafo coincide ovviamente con il numero m di terne nel file. Come numero n di vertici si prende invece il massimo intero che compare come u_i o v_i (cioè come indicativo di un vertice). Se poi un numero minore di n non compare, si assume che il corrispondente nodo sia isolato.**

2.16 **Ad esempio, il grafo:**

2 3 2
5 2 4
3 4 1
5 1 2
1 3 4
7 2 4

2.17 **ha 7 vertici (perché 7 è il massimo valore che compare nelle prime due posizioni delle terne) e i seguenti archi: (2,3), (5,2), (3,4), (5,1), (1,3), (7,2). Il vertice 6 è dunque isolato. In questo caso, inoltre, i pesi sono tutti interi e positivi.**

```
In [ ]: def readgraph(fn):
            '''Legge il grafo da file. Ogni riga deve essere composta da tre numeri:
```

```
        i primi due rappresentano i nodi (estremi dell'arco) mentre il terzo
        rappresenta il peso.'''
    E = []
    with open(fn) as f:
        for l in f:                          # l è una riga del file, letta come stringa
            tokens = l.strip().split(' ') # strip() elimina caratteri "sporchi" a fine
                                          # split restituisce una lista di token (defi
            u = int(tokens[0])            # Il primo token rappresenta un vertice (dev
            v = int(tokens[1])            # Idem per il secondo
            w = float(tokens[2])          # Il terzo token rappresenta il peo (deve es
            e = edge((u,v),w)
            E.append(e)
    return E
```

2.18 Mettiamo insieme ciò che abbiamo fatto finora

```
In [ ]: if 'edge' in dir():
            del(edge)

        class edge(tuple):
            '''Classe che rappresenta archi non orientati pesati, v2.0.
            Eredita da tuple.
               '''
            def __new__(cls,e,w):
                '''Ridefiniamo il metodo __new__ perché abbiamo bisogno di specificare
                   un parametro addizionale e non solo la coppia e=(u,v)'''
                edge = super().__new__(cls,e)
                edge.w = w
                return edge

            def __str__(self):
                '''Definisce la rappresentazione di oggetti edge. Usata da print() e str().
                   Occhio alle differenze con le versioni precedenti.'''
                s = super().__str__()+": "+str(self.w)
                # s = tuple.__str__(self)     # Alternativa, meno in linea con Python3
                return s

            def __lt__(self,other):
                '''Definire __lt__ forza un nuovo comportamento di <.
                   Metodo che confronta l'arco self con other, sulla base
                   prima del peso e poi dell'ordine lessicografico.
                   Anche qui, occhio alle differenze con le versioni precedenti'''
                ws = self.w
                wo = other.w
                ms = min(self)
                mo = min(other)
                if ws<wo or (ws==wo and ms<mo) or \
                    (ws==wo and ms==mo and max(self)<max(other)):
```

```
            return True
        else:
            return False

def readgraph(fn):
    '''Legge il grafo da file. Ogni riga deve essere composta da tre numeri:
        i primi due rappresentano i nodi (estremi dell'arco) mentre il terzo
        rappresenta il peso.'''
    E = []
    with open(fn) as f:
        for l in f:                          # l è una riga del file, letta come stringa
            tokens = l.strip().split(' ') # strip() elimina caratteri "sporchi" a fine
                                          # split restituisce una lista di token (defi
            u = int(tokens[0])            # Il primo token rappresenta un vertice (dev
            v = int(tokens[1])            # Idem per il secondo
            w = float(tokens[2])          # Il terzo token rappresenta il peo (deve es
            e = edge((u,v),w)
            E.append(e)
    return E

def edgesort(E):
    '''Usa insertion sort per ordinare una lista di archi'''
    n = len(E)
    for i in range(1,n):
        e = E[i]
        j = i-1
        while j>=0 and e<E[j]:
            E[j+1] = E[j]
            j = j-1
        E[j+1] = e

if __name__=='__main__':
    E = readgraph('graph1.txt')
    print("Prima:")
    for e in E:
        print(e)
    edgesort(E)
    print("Dopo")
    for e in E:
        print(e)
```

2.19 Definiamo ora una classe *grafo*, che supporremo memorizzato nel file *simple-graph.py* insieme alla classe *edge*

```
In [ ]: if 'edge' in dir():
            del(edge)
        if 'graph' in dir():
            del(graph)
```

```
class edge(tuple):
    '''Classe che rappresenta archi non orientati pesati, v2.0.
    Eredita da tuple.
       '''
    def __new__(cls,e,w):
        '''Ridefiniamo il metodo __new__ perché abbiamo bisogno di specificare
           un parametro addizionale e non solo la coppia e=(u,v)'''
        edge = super().__new__(cls,e)
        edge.w = w
        return edge

    def __str__(self):
        '''Definisce la rappresentazione di oggetti edge. Usata da print() e str().
           Occhio alle differenze con le versioni precedenti.'''
        s = super().__str__()+": "+str(self.w)
        # s = tuple.__str__(self)     # Alternativa, meno in linea con Python3
        return s

    def __lt__(self,other):
        '''Definire __lt__ forza un nuovo comportamento di <.
           Metodo che confronta l'arco self con other, sulla base
           prima del peso e poi dell'ordine lessicografico.
           Anche qui, occhio alle differenze con le versioni precedenti'''
        ws = self.w
        wo = other.w
        ms = min(self)
        mo = min(other)
        if ws<wo or (ws==wo and ms<mo) or \
            (ws==wo and ms==mo and max(self)<max(other)):
            return True
        else:
            return False

class graph:
    '''Classe che descrive un grafo pesato con vertex set V={1,2,...,n}
       Il grafo viene inizializzato (come in precedenza) leggendo
       la lista degli archi e i relativi pesi da file.
       Vengono poi creati il vertex set V e un dizionario A che
       rappresenta le liste di adiacenza. La classe usa gli attributi "privati"
       __E, __A, __V ed __n definendo corrispondenti proprietà per controllarne
       gli accessi.'''
    def __init__(self,fn):
        '''Inizializza il grafo leggendo la lista degli archi e costruendo l'insieme d
           e il dizionario che rappresenta le liste di adiacenza'''
        self.__E = []                           # Lista egli archi
        self.__V = set()                        # Insieme dei vertici
        self.__A = {}                           # Dizionario per le adiacenze dei vert
```

```
        self.__n = 0                              # self.n memorizza sempre il massimo v
        with open(fn) as f:
            for l in f:                           # l è una riga del file, letta come stri
                tokens = l.strip().split(' ') # strip() elimina caratteri "sporchi" a
                                                  # split restituisce una lista di token (
                u = int(tokens[0])                # Il primo token rappresenta un vertice
                v = int(tokens[1])                # Idem per il secondo
                w = float(tokens[2])              # Il terzo token rappresenta il peo (dev
                e = edge((u,v),w)
                self.__E.append(e)
                M = max(u,v)
                m = min(u,v)
                if M > self.__n:                  # Questa condizione implica che M non
                    for i in range(self.__n+1,M+1):    # Vengono quindi inseriti tutti
                        self.__V.add(i)                # maggiore di n e non maggiore
                        self.__A[i] = []
                    self.__n = M                       # Si aggiorna il valore di n
                self.__A[M].append(m)                  # Si aggiornano le liste di adi
                self.__A[m].append(M)
        super().__setattr__('E',self.__E)
        super().__setattr__('V',self.__V)
        super().__setattr__('A',self.__A)
        super().__setattr__('n',self.__n)

    @property
    def A(self):
        return self.__A

    @A.setter
    def A(self,x):
        return

    @property
    def E(self):
        return self.__E

    @E.setter
    def E(self,x):
        return

    @property
    def V(self):
        return self.__V

    @V.setter
    def V(self,x):
        return
```

```
    @property
    def n(self):
        return self.__n

    @n.setter
    def n(self,x):
        return
```

2.20 Nuova versione del programma

```
In [ ]: #Definizione o importazione delle classi edge e graph

    def edgesort(E):
        '''Usa insertion sort per ordinare una lista di archi'''
        n = len(E)
        for i in range(1,n):
            e = E[i]
            j = i-1
            while j>=0 and e<E[j]:
                E[j+1] = E[j]
                j = j-1
            E[j+1] = e

    G = graph('graph1.txt')   # graph1.txt contiene l'elenco degli archi

    for e in G.E:
        print(e)

    edgesort(G.E)

    print()

    for e in G.E:
        print(e)
```

2.21 Dobbiamo ora definire una classe che implementa la struttura dati Union-Find

```
In [ ]: class unionfind:
            '''Classe che implementa una struttura dati per insiemi disgiunti (union-find)
            secondo lo schema quick-union e find con compressione del cammino.
            I dati sono mantenuti in due liste: (1) la lista __p, che memorizza i parent di og
            (2) la lista __d che memorizza la dimensione dei vari sottoinsiemi (più correttame
           __d[x] memorizza il numero di elementi che aveva il sotto-insieme cui appartiene x
           in cui questo è stato per l'ultima volta il rappresentante del suo sotto-insieme)''
            def __init__(self, n):
                self.__p = ['unused'] + list(range(1,n+1)) # Se _p[i]==i, allora i è rappresen
                self.__d = ['unused'] + [1] * n            # _d[i] è il peso (num. di elementi
```

```
            def find(self, x):
                '''Calcola il rappresentante del sotto-insieme cui appartiene i.
                   Usa la path-compression per ridurre l'altezza dell'albero che implementa
                   il sotto-insieme.'''
                j = x
                while (j != self.__p[j]):
                    self.__p[j] = self.__p[self.__p[j]]   # _p[j] ora punta al "nonno" di j
                    j = self.__p[j]                       # ... e j risale al nonno
                return j

            def union(self,x,y):
                '''La union fa puntare il sotto-insieme di peso minore al sotto-insieme di pes
                xr = self.find(x)
                yr = self.find(y)
                if xr != yr:
                    if self.__d[xr]<self.__d[yr]:
                        self.__p[xr] = yr
                        self.__d[yr] += self.__d[xr]
                    else:
                        self.__p[yr] = xr
                        self.__d[xr] += self.__d[yr]
```

```
In [ ]: U = unionfind(10)
        print(U._unionfind__p)
        print(U._unionfind__d)
```

```
In [ ]: U.union(1,2)
        U.union(3,9)
        U.union(1,9)
        U.find(2)
```

```
In [ ]: print(U._unionfind__p)
        print(U._unionfind__d)
```

2.22 Il programma finalmente completo

```
In [ ]: class edge(tuple):
            '''Classe che rappresenta archi non orientati pesati, v2.0.
            Eredita da tuple.
               '''
            def __new__(cls,e,w):
                '''Ridefiniamo il metodo __new__ perché abbiamo bisogno di specificare
                   un parametro addizionale e non solo la coppia e=(u,v)'''
                edge = super().__new__(cls,e)
                edge.w = w
                return edge

            def __str__(self):
```

```
        '''Definisce la rappresentazione di oggetti edge. Usata da print() e str().
           Occhio alle differenze con le versioni precedenti.'''
        s = super().__str__()+": "+str(self.w)
        # s = tuple.__str__(self)     # Alternativa, meno in linea con Python3
        return s

    def __lt__(self,other):
        '''Definire __lt__ forza un nuovo comportamento di <.
           Metodo che confronta l'arco self con other, sulla base
           prima del peso e poi dell'ordine lessicografico.
           Anche qui, occhio alle differenze con le versioni precedenti'''
        ws = self.w
        wo = other.w
        ms = min(self)
        mo = min(other)
        if ws<wo or (ws==wo and ms<mo) or \
            (ws==wo and ms==mo and max(self)<max(other)):
            return True
        else:
            return False

class graph:
    '''Classe che descrive un grafo pesato con vertex set V={1,2,...,n}
       Il grafo viene inizializzato (come in precedenza) leggendo
       la lista degli archi e i relativi pesi da file.
       Vengono poi creati il vertex set V e un dizionario A che
       rappresenta le liste di adiacenza. La classe usa gli attributi "privati"
       __E, __A, __V ed __n definendo corrispondenti proprietà per controllarne
       gli accessi.'''
    def __init__(self,fn):
        '''Inizializza il grafo leggendo la lista degli archi e costruendo l'insieme d
           e il dizionario che rappresenta le liste di adiacenza'''
        self.__E = []                           # Lista egli archi
        self.__V = set()                        # Insieme dei vertici
        self.__A = {}                           # Dizionario per le adiacenze dei vert
        self.__n = 0                            # self n memorizza sempre il massimo v
        with open(fn) as f:
            for l in f:                       # l è una riga del file, letta come stri
                tokens = l.strip().split(' ') # strip() elimina caratteri "sporchi" a
                                              # split restituisce una lista di token (
                u = int(tokens[0])            # Il primo token rappresenta un vertice
                v = int(tokens[1])            # Idem per il secondo
                w = float(tokens[2])          # Il terzo token rappresenta il peo (dev
                e = edge((u,v),w)
                self.__E.append(e)
                M = max(u,v)
                m = min(u,v)
                if M > self.__n:                # Questa condizione implica che M non
```

```
                    for i in range(self.__n+1,M+1):     # Vengono quindi inseriti tutti
                        self.__V.add(i)                 # maggiore di n e non maggiore
                        self.__A[i] = []
                    self.__n = M                        # Si aggiorna il valore di n
                self.__A[M].append(m)                   # Si aggiornano le liste di adi
                self.__A[m].append(M)
            super().__setattr__('E',self.__E)
            super().__setattr__('V',self.__V)
            super().__setattr__('A',self.__A)
            super().__setattr__('n',self.__n)

        @property
        def A(self):
            return self.__A

        @A.setter
        def A(self,x):
            return

        @property
        def E(self):
            return self.__E

        @E.setter
        def E(self,x):
            return

        @property
        def V(self):
            return self.__V

        @V.setter
        def V(self,x):
            return

        @property
        def n(self):
            return self.__n

        @n.setter
        def n(self,x):
            return

    class unionfind:
        '''Classe che implementa una struttura dati per insiemi disgiunti (union-find)
        secondo lo schema quick-union e find con compressione del cammino.
        I dati sono mantenuti in due liste: (1) la lista __p, che memorizza i parent di og
        (2) la lista __d che memorizza la dimensione dei vari sottoinsiemi (più correttame
```

```
    __d[x] memorizza il numero di elementi che aveva il sotto-insieme cui appartiene x
    in cui questo è stato per l'ultima volta il rappresentante del suo sotto-insieme)''
    def __init__(self, n):
        self.__p = ['unused'] + list(range(1,n+1)) # Se _p[i]==i, allora i è rappresen
        self.__d = ['unused'] + [1] * n            # _d[i] è il peso (num. di elementi

    def find(self, x):
        '''Calcola il rappresentante del sotto-insieme cui appartiene i.
           Usa la path-compression per ridurre l'altezza dell'albero che implementa
           il sotto-insieme.'''
        j = x
        while (j != self.__p[j]):
            self.__p[j] = self.__p[self.__p[j]]   # _p[j] ora punta al "nonno" di j
            j = self.__p[j]                       # ... e j risale al nonno
        return j

    def union(self,x,y):
        '''La union fa puntare il sotto-insieme di peso minore al sotto-insieme di pes
        xr = self.find(x)
        yr = self.find(y)
        if xr != yr:
            if self.__d[xr]<self.__d[yr]:
                self.__p[xr] = yr
                self.__d[yr] += self.__d[xr]
            else:
                self.__p[yr] = xr
                self.__d[xr] += self.__d[yr]

def edgesort(E):
    '''Usa insertion sort per ordinare una lista di archi'''
    n = len(E)
    for i in range(1,n):
        e = E[i]
        j = i-1
        while j>=0 and e<E[j]:
            E[j+1] = E[j]
            j = j-1
        E[j+1] = e

G = graph('graph1.txt')   # graph1.txt contiene l'elenco degli archi

print("Grafo di input")
for e in G.E:
    print(e)

edgesort(G.E)
```

```
U = unionfind(G.n)

count = 0
MST = []
for e in G.E:
    u = e[0]
    v = e[1]
    if U.find(u) != U.find(v):
        U.union(u,v)
        MST.append(e)
        count += 1
        if count == G.n-1:
            break

print("\nMinimum Spanning Tree")
for e in MST:
    print(e)
```

2.23 Come script invocabile da shell:

```
In [ ]: #!/usr/bin/env python3
        # -*- coding: utf-8 -*-

        from simplegraph import graph
        from unionfind import unionfind

        def edgesort(E):
            '''Usa insertion sort per ordinare una lista di archi'''
            n = len(E)
            for i in range(1,n):
                e = E[i]
                j = i-1
                while j>=0 and e<E[j]:
                    E[j+1] = E[j]
                    j = j-1
                E[j+1] = e

        def main():


            G = graph('graph1.txt')   # graph1.txt contiene l'elenco degli archi

            print("Grafo di input")
            for e in G.E:
                print(e)

            edgesort(G.E)
```

```
    U = unionfind(G.n)

    count = 0
    MST = []
    for e in G.E:
        u = e[0]
        v = e[1]
        if U.find(u) != U.find(v):
            U.union(u,v)
            MST.append(e)
            count += 1
            if count == G.n-1:
                break

    print("\nMinimum Spanning Tree")
    for e in MST:
        print(e)


if __name__=='__main__':
    main()
```