INDICE

tuple (iterabile, non-mutable)

Le tuple sono molto simili alle liste ma, a differenza di queste ultime NON SONO MODIFICABILI IN-PLACE (sono non-mutable).

Ha senso utilizzare le tuple quando vogliamo essere sicuri che il contenuto di una sequenza non vari durante la sua esistenza!

Caratteristiche in comune con le liste

- possono contenere oggetti di tipo eterogeneo
- Supportano gli stessi meccanismi di indicizzazione delle liste (in particolare supportano gli slice)
- Sono iterabili

Principali differenze rispetto alle liste

- Non sono modificabili in-place (e pertanto non si possono aggiungere o togliere elementi da una toupla (ma e' possibile definire una toupla a partire dagli elementi di un'altra))
- NON supportano un meccanismo analogo alla list comprehension anche se e' possibile creare una tupla con sintassi simile

Creare una tupla

parentesi tonde, elemento per elemento, separati da virgola

In [58]:
```
#definisco una tupla usando le parentesi tonde
#NB: elementi eterogenei
t1 = (1,'ciao',65.23,[1,2,3])
```

le parentesi tonde sono opzionali, una serie di elementi separati da virgola e' interpretata come tupla

In [59]:
```
#posso definire una toupla anche senza le parentesi tonde,
#specificandone gli elementi separati da virgola
t2 = 1,'ciao',65.23,[1,2,3]
print(t1)
```

```
(1, 'ciao', 65.23, [1, 2, 3])
```

a partire da altre tuple

uso gli operatori + e *

In [60]:
```
t2 = t1*3
print(t2)
```

```
(1, 'ciao', 65.23, [1, 2, 3], 1, 'ciao', 65.23, [1, 2, 3], 1, 'ciao', 65.23, [1, 2, 3])
```

Posso anche utilizzare gli slice per estrarre pezzi di tupla (vedi sotto *)

Accedere agli elementi di una tupla

In [61]:
```
#stampo il secondo elemento
print(t1[1])
```

```
ciao
```

In [62]:
```
#stampo il secondo ed il terzo elemento, estratti con uno slice
print(t1[1:3])
```

```
('ciao', 65.23)
```

lo slice ci restituisce una tupla!
(*) Di conseguenza posso usare lo slicing e gli operatori + e * per creare altre tuple:

In [63]:
```
t2[0:3]+2*t1[0:1] #OK
```

```
t2[0:3]+2*t1[0]  #KO
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-63-5a5711462749> in <module>()
      1 t2[0:3]+2*t1[0:1] #OK
----> 2 t2[0:3]+2*t1[0]  #KO

TypeError: can only concatenate tuple (not "int") to tuple
```

Iterare su una tupla

In [64]:
```
#itero sulla toupla
for elem in t1:
    print(elem)
```

```
1
ciao
65.23
[1, 2, 3]
```

Modifica del contenuto degli elementi di una tupla

Si è gia' detto che non posso assegnare un nuovo valore ad un elemento della toupla

In [65]:
```
t1 = (1,'ciao',65.23,[1,2,3])
#nel caso in cui l'elemento in questione è non mutable (t1[0], int)
#t1[0] = 0 # ERRORE
#ma anche quando e' mutable (t1[3], list)
#t1[3] = [] # ERRORE
```

tuttavia posso modificare 'in-place' un elemento mutable contenuto in una tupla:

In [66]:
```
print(t1)
idtuplaold = id(t1)
idlistaold = id(t1[3])
#NB: il quarto elemento della toupla e' una lista e quindi puo' essere modificato!!
t1[3][1]=10*t1[3][1]
print(t1)
idtuplanew=id(t1)
idlistanew=id(t1[3])
print(idtuplaold==idtuplanew)
print(idlistaold==idlistanew)
```

```
(1, 'ciao', 65.23, [1, 2, 3])
(1, 'ciao', 65.23, [1, 20, 3])
True
True
```

non solo e' possibile modificare gli elementi della lista ma e' anche possibile variare il numero di elementi, purche' le modifiche avvengano in-place

In [67]:
```
#NB anche se t1[3]=[] non e' permesso, questo invece lo posso fare!!!
del(t1[3][:])
print(t1)
t1[3].append(1)
```

```
print(t1)
```

```
(1, 'ciao', 65.23, [])
(1, 'ciao', 65.23, [1])
```

Le stringhe (iterabile, non-mutable)

- str in py3k doc

Le stringhe possono essere pensate come delle `touple di soli caratteri`. Quindi:
- sono immutabili
- supportano l'indicizzazione
- supportano gli slice
- supportano gli operatori + e * che funzionano come per liste e touple.
- supportano l'iterazione

Creazione di stringhe

Literals

Si puo' definire una nuova stringa con singoli apici ' :

In [68]:
```
str1 = 'allows embedded "double" quotes'
print(str1)
```

```
allows embedded "double" quotes
```

Con doppi apici " :

In [69]:
```
str2 = "allows embedded 'single' quotes"
print(str2)
```

```
allows embedded 'single' quotes
```

Con tripli apici ''' o """ :

In [70]:
```
str3 = """questa stringa si
puo' estendere su piu' righe e puo' includere
apici singoli: ', doppi apici " e altro """
print(str3)
```

```
questa stringa si
puo' estendere su piu' righe e puo' includere
apici singoli: ', doppi apici " e altro
```

La funzione str

La funzione str converte in stringa (quando e' possibile) un generico altro oggetto.
Di fatto permette la conversione di tipo a stringa.

In [71]:
```
#da intero
si = str(234)
si
```

Out[71]:
```
'234'
```

In [72]:
```
#da float
sf = str(1e-4)
sf
```

Out[72]: '0.0001'

In [73]:
```
#da liste
sl = str([1,2,3,4,5])
sl
```

Out[73]: '[1, 2, 3, 4, 5]'

In [74]:
```
#da complessi
sc = str(1+1j)
sc
```

Out[74]: '(1+1j)'

funziona con tutti gli oggetti forniti di metodo __str__()

In [75]:
```
'__str__' in dir(int)
```

Out[75]: True

Come risultato di operazioni

Indicizzazione

L'indicizzazione funziona come per le liste, con le parentesi quadre

In [76]:
```
#indicizzazione
str3[3]
```

Out[76]: 's'

Slices

In [77]:
```
#slice
str3[7:14]
```

Out[77]: 'stringa'

In [78]:
```
str3[7:25]
```

Out[78]: "stringa si \npuo' e"

Escape sequences

Notare la notazione /n che fa parte delle cosiddette 'escape sequences':

Escape Sequence	Meaning
`\newline`	Backslash and newline ignored
`\\`	Backslash (`\`)
`\'`	Single quote (`'`)
`\"`	Double quote (`"`)
`\a`	ASCII Bell (BEL)
`\b`	ASCII Backspace (BS)
`\f`	ASCII Formfeed (FF)
`\n`	ASCII Linefeed (LF)
`\r`	ASCII Carriage Return (CR)
`\t`	ASCII Horizontal Tab (TAB)
`\v`	ASCII Vertical Tab (VT)
`\ooo`	Character with octal value *ooo*
`\xhh`	Character with hex value *hh*

Funzioni membro

Oltre a cio' le stringhe sono un tipo di dati che e' caratterizzato da molte funzioni membro, specifiche per il testo:

In [1]:
```
s = 'questa è una stringa minuscola'
S = s.upper()
S = S.replace('MINUSCOLA','MAIUSCOLA')
print(s)
print(S)
```

```
questa è una stringa minuscola
QUESTA È UNA STRINGA MAIUSCOLA
```

In [2]:
```
s.count('s'),s.count('S')
```

Out[2]:
```
(3, 0)
```

Per verificare se la stringa comprende una parola usare l'operatore in:

In [3]:
```
'stringa' in s , 'Stringa' in s
```

Out[3]:
```
(True, False)
```

Per trovare l'inizio della prima occorrenza di una sottostringa:

In [4]:
```
s.find('stringa'),s.find('Stringa')
```

Out[4]: (13, -1)

In [5]:
```
s[s.find('stringa'):]
```

Out[5]: 'stringa minuscola'

Formattazione di stringhe (base)

Ci sono due metodi di formattazione di stringhe in Python:

I. Quello che usa la funzione `format`
II. Quello che usa l'operatore %

Noi ci concentreremo sul primo, non perche' sia migliore, ma perche' più 'caratteristico' di Python e più moderno e si rimanda ai riferimenti per la seconda soluzione (che ricorda molto la funzione printf in linguaggio C).
Partiamo subito con degli esempi:

Associazione segnaposto {} nella stinga ai parametri passati a format

- Definisco una stringa in cui ci sono dei segnaposto
- I segnaposto sono riempiti con i parametri passati alla funzione format, che e' una funzione membro della classe stringa
- Definiamo nel seguito come si puo' associare un segnaposto al valore del parametro che sara' rappresentato nella stringa restituita da format

In [6]:
```
#ASSOCIAZIONE POSIZIONE
'{},{},{}'.format('a',1.0,100)
#la coppia di parentesi graffe delimitano il segnaposto
#se sono vuote inserisco gli argomenti di format nell'ordine in cui sono elencati
```

Out[6]: 'a,1.0,100'

In [7]:
```
#ASSOCIAZIONE INDICE
'{2},{0},{1}'.format('a',1.0,100)
#se dentro i segnaposto ci metto gli indici che individuano gli argomenti
#0 il primo, 1 il seconde etc.
#posso cambiare l'ordinamento dei parametri nei segnaposto:
```

Out[7]: '100,a,1.0'

In [8]:
```
#oppure posso rferirmi più volte allo stesso parametro:
'{2},{2},{1}'.format('a',1.0,100)
```

Out[8]: '100,100,1.0'

In [9]:
```
#ASSOCIAZIONE PER NOME
'{c},{a},{b}'.format(a= 'a',b= 1.0,c= 100)
```

Out[9]: '100,a,1.0'

In [10]:
```
#NB l'associazione del nome deve avvenire all'interno della format
#questo non funzione
#a = 10
#'{a}'.format(a)

#questo invece si', ma attenzione a cosa succede!!
```

```
a=10
print('{a}'.format(a='pippo'))
print(a)
```

```
pippo
10
```

In [11]:
```
#ASSOCIAZIONE PER INDICE+ATTRIBUTO
c = 1+2j
'parte reale: {0.real}, parte immaginaria: {0.imag}'.format(c)
#NB: non funziona con le funzioni membro
```

Out[11]:
```
'parte reale: 1.0, parte immaginaria: 2.0'
```

In [12]:
```
#ASSOCIAZIONE PER INDICE + INDICE
l=[1,'due',3.0]
'primo elemento: {0[0]}, secondo: {0[1]}, terzo: {0[2]}'.format(l)
```

Out[12]:
```
'primo elemento: 1, secondo: due, terzo: 3.0'
```

Ci sono altri tipi di associazioni ma si basano su elementi che ancora non abbiamo visto (ad esempio sui dizionari).

Formattazione dei campi nei segnaposto

La formattazione dei vari campi e' controllabile grazie ad un vero e proprio mini linguaggio di cui vedremo solo qualche esempio:

In [13]:
```
#float con segno numero minimo di caratteri e precisione 4
'{0:+.4f}'.format(3.123)
```

Out[13]:
```
'+3.1230'
```

In [14]:
```
#float con segno, 10 caratteri e precisione 1
'{0:+10.1f}'.format(3.123)
```

Out[14]:
```
'      +3.1'
```

In [15]:
```
#carattere di riempimento'X', allineato a destra, float con segno, 10 caratteri e
precisione 1
'{0:X>+10.1f}'.format(3.123)
```

Out[15]:
```
'XXXXXX+3.1'
```

Vedi anche format examples

Dietro le quinte: stringhe ed unicode in Python 3.X

- Unicode HOWTO in py3k doc
- love-hotels-and-unicode

In prima istanza le stringhe possono essere utilizzate senza curarsi di come sono rappresentate nella memoria del calcolatore (o come scritte sul disco).
Tuttavia questo e' un aspetto di fondamentale importanza e lo affronteremo qui, almeno a livello di cenni, rimandando ai link esterni per approfondimenti.

Le stringhe sono rappresentate da sequenze di caratteri UNICODE.

La rappresentazione unicode dei caratteri si basa su due aspetti fondamentali:

I. L'associazione ad ogni carattere di un codice univoco (*code point*)
II. L'associazione ad ogni code point di una rappresentazione binaria (*encoding)*

Code point e caratteri

- Il consorzio UNICODE si occupa di associare ad ogni carattere un codice univoco detto *code point*.
- Il code point e' un numero intero che va da 0 a 1 114 112 (decimale) (al settembre 2012 i caratteri definiti dal consorzio e associati ad un code-point sono 110182).
- Il code point e' spesso rappresentato in base esadecimale (il primo codice fuori range e' 0x110000).

Il code point non ci dice come stampare un carattere (questo aspetto e' definito dal font utilizzato).
La a minuscola ha code point U+0061 indipendentemente da come e' resa:

Tra gli oltre 100 mila caratteri definiti dalla specifica unicode 6.2 ce ne sono di molti tipi:

oppure il raffinato:

Python e code point

chr

La funzione builtin chr accetta come parametro il code point (intero) e restituisce il carattere ad esso associato (segnala un errore se l'intero e' fuori dal range dei code point validi)

In [16]:
```
c1 = chr(0x61)
c1
```

Out[16]:
```
'a'
```

In [17]:
```
c2 = chr(0x0393)
c2
```

Out[17]:
```
'Γ'
```

Posso anche creare un carattere noto il suo nome UNICODE:

In [18]:
```
c3 = "\N{GREEK CAPITAL LETTER GAMMA}"
c3
```

Out[18]:
```
'Γ'
```

Fuori range:

In [19]:
```
#chr(2000000)
```

Come sopra ma un po'più elegante:

In [20]:
```
for code in range(1000000,2000000):
    try: #io ci provo ...
        chr(code)
    except: #se c'e' un errore
        print("{} e' l'ultimo codice valido".format(code-1))
        print('{} e' fuori dal range dei codici validi'.format(code))
        break
```

```
  File "<ipython-input-20-5f1f0b96e82e>", line 6
    print('{} e' fuori dal range dei codici validi'.format(code))
                      ^
SyntaxError: invalid syntax
```

ord

La funzione builtin ord accetta come parametro un carattere e restituisce il code point ad esso associato:

In [21]:
```
c1 = 'a'
c1,ord(c1),hex(ord(c1))
```

Out[21]:
```
('a', 97, '0x61')
```

In [22]:
```
c2 = chr(0x0393)
```

```
c3 = "\N{GREEK CAPITAL LETTER GAMMA}"
ord(c2), hex(ord(c2)) , ord(c2) == ord(c3)
```

Out[22]: (915, '0x393', True)

In [23]:
```
[hex(b) for b in chr(0x0393).encode('utf-8')]
```

Out[23]: ['0xce', '0x93']

Encoding

La rappresentazione binaria del code point e' detta *codifica (encoding)* ed e' il secondo aspetto che affrontiamo

- L'encoding e' il procedimento che permette di associare ad un code-point unicode una sequenza di byte.
- La progettazione di un encoding deve tenere conto di molti fattori come l'occupazione di memoria e l'efficienza delle varie operazioni da eseguire sulle stringhe.

Quanti bit sono necessari per rappresentare rappresentare i numeri interi da zero fino a 0x110000 (il code point + grande al sett-2012)?

In [24]:
```
print(bin(0x110000))
print(len(bin(0x110000)[2:]))#non considero i caratteri 0x
```

```
0b100010000000000000000
21
```

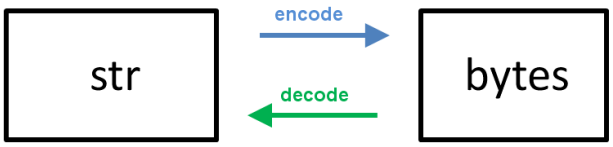
Un altro modo per verificarlo:

In [25]:
```
import math
math.log(1114111,2)
```

Out[25]: 20.087461546321563

Ci vogliono almeno 21 bit, quindi minimo 3 byte (se vogliamo utilizzare una codifica a lunghezza fissa), ovviamente se vogliamo rappresentare tutti i possibili code-point.

Tuttavia non tutte le codifiche mirano a rappresentare tutti i caratteri gestiti da unicode
Ad esempio la codifica ASCII mappa solo i primi 128 code point in parole di 7 bit.
Ma vedremo questo fra poco perche' per lavorare con gli encoding e' necessario prima introdurre il tipo di dati denominato `bytes`.



bytes

- Il tipo bytes corrisponde ad una toupla di interi ciascuno dei quali puo' assumere il valore massimo pari a 255 (esiste anche la versione mutable che si chiama `bytearray`).
- Il tipo bytes puo' anche essere visto come una stringa di soli caratteri ascii, tanto e' che supporta molte delle funzionalita' delle stringhe (vedere dir(b'1'))

Per creare i bytes

- funzione builtin bytes
- byte literals (cioe' uso una stringa ascii preceduta dalla lettera b)
- codifica di una stringa secondo uno specifico 'encoding' (uso il metodo encode della stringa)

In [26]:
```
bytes([255])
bytes(2)
```

Out[26]:
```
b'\x00\x00'
```

In [27]:
```
a = bytes([49]) #funzione builtin bytes
b = b'1'        #bytes literal
c = '1'.encode('ascii') #codifica di una stringa secondo uno specifico 'encoding'

#stampo la versione 'stampabile' dei bytes
#cioe' stampo la stringa ascii corrispondente
print(a,b,c)
#stampo sia la versione stampabile
#sia l'intero corrispondente
print(a,a[0],bin(a[0]),hex(a[0]))
print(b,b[0],bin(b[0]),hex(b[0]))
print(c,c[0],bin(c[0]),hex(c[0]))
```

```
b'1' b'1' b'1'
b'1' 49 0b110001 0x31
b'1' 49 0b110001 0x31
b'1' 49 0b110001 0x31
```

operazioni sui bytes

- bytes in py3k doc

Gli operatori aritmetici e le funzioni membro dei bytes lavorano come per le stringhe

In [28]:
```
print(a+b) #concatenamento
print(a*5) #ripetizione
# ....      etc etc
```

```
b'11'
b'11111'
```

Se si vuole operare sul valore dei singoli byte e' necessario accedervi singolarmente, ad esempio mediante iterazione:

In [29]:
```
d = bytes([bi for i,bi in enumerate(range(5))])
print(d)
```

```
b'\x00\x01\x02\x03\x04'
```

ricordarsi comunque che non e' possiie eseguire operazioni 'in-place' (per questo ci vogliono i bytearray)

In [30]:
```
#d[1]=d[1]+1 #errore!!!
```

In [31]:
```
d = bytearray([bi for i,bi in enumerate(range(5))])
d[1]=d[1]+1
print(d)
```

```
bytearray(b'\x00\x02\x02\x03\x04')
```

stringhe, bytes ed encoding

unicode-utf8-table

conversione da bytes a stringa (decodifica)

- funzione builtin str
- metodo decode dei bytes

funzione builtin str

La funzione builtin str oltre a permettere la conversione in stringa di vari tipi di oggetto permette anche di provare vari encoding. Se viene specificato il parametro 'encoding' restituisce la rappresentazione come stringa della sequenza di bytes passata come primo argomento, secondo l'encoding specificato.
Come sempre meglio un esempio di mille parole:

In [32]:
```
a = bytes([1])
b = b'1'
str(a,encoding='ascii') , str(b,encoding='ascii')
```

Out[32]:
```
('\x01', '1')
```

In [33]:
```
#str(b'\xe4',encoding='latin-1')
```

In [34]:
```
#tutti i caratteri ascii
nch=128
#str(bytes(range(nch)),encoding='ascii')
```

In [35]:
```
nch=256 #provare con 257
#str(bytes(range(256)),encoding='latin-1')
```

funzione membro decode dei bytes

In [36]:
```
#bytes litteral diversi, stessa stringa con diversi encoding
s1 = b'perch\xe9'.decode('latin-1')
s2 = b'perch\xc3\xa9'.decode('utf-8')
print(s1)
print(type(s1))
```

```
print(s1 == s2)
```

```
perché
<class 'str'>
True
```

In [37]:
```
#bytes uguali -> stringhe diverse con diversi encoding
#print(bytes([195,169]).decode('latin-1'))
#print(bytes([195,169]).decode('utf-8'))
```

conversione da stringa a bytes (codifica)

- metodi encode delle stringhe
- funzione builtin bytes (introdotta quando si e' parlato del tipo bytes)

funzione membro encode delle stringhe

Per 'giocare' un po con gli encoding possiamo usare le funzioni `encode` del tipo str

In [38]:
```
stringa = 'perché'
print(stringa.encode('latin-1'))
print(stringa.encode('utf-8'))
print("come si può vedere ottengo due diverse rappresentazioni della stessa stringa")
```

```
b'perch\xe9'
b'perch\xc3\xa9'
come si può vedere ottengo due diverse rappresentazioni della stessa stringa
```

stampo i bytes come interi:

In [39]:
```
stringa = "perché"
print([b for b in stringa.encode('latin-1')])
print([b for b in stringa.encode('utf-8')])
```

```
[112, 101, 114, 99, 104, 233]
[112, 101, 114, 99, 104, 195, 169]
```

funzione builtin bytes

In [41]:
```
b1=bytes('perché',encoding='latin-1')
b2=bytes('perché',encoding='utf-8')
print(b1)
print(b2)
```

```
b'perch\xe9'
b'perch\xc3\xa9'
```

e con questo, vi saluto:


WAVING HAND SIGN
U+1F44B