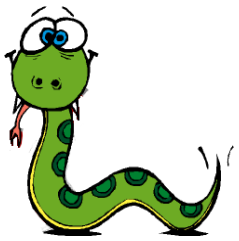
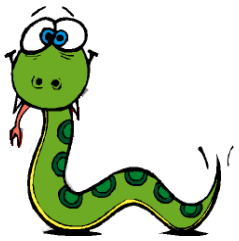
Tecnologie dell'Informazione e della Comunicazione



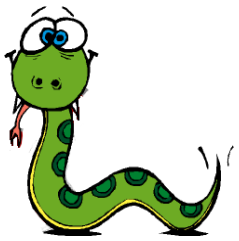
Capitolo 7, 8, 9
Strutture dati

Prof. Mauro Gaspari: gaspari@cs.unibo.it



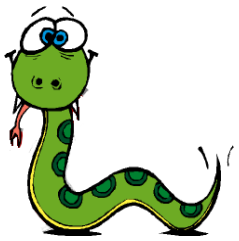
Un tipo di dato strutturato

- Tipi visti fino ad ora: int, float, string
- Le stringhe sono diversi dagli altri tipi perche' sono composte da pezzi piu' piccoli: i caratteri.
- I tipi di dato che hanno queste caratteristiche si dicono **strutturati o composti** (= **compound data types**).
- I tipi int e float si dicono invece **scalari**.



Tipologie di Strutture dati

- Stringhe
- Liste
- Tuple
- Dizionari



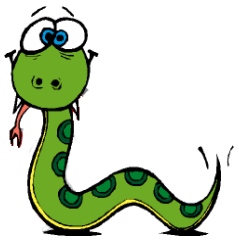
Caratteristiche tipi strutturati

- Di solito i tipi di dato strutturati si possono considerare come un oggetto unico oppure e' possibile accedere alle loro parti.
- Ad esempio:

```
>>> fruit = "banana"
>>> letter = fruit[1]
>>> print letter
a
```

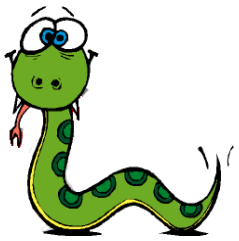
- Perche' “a”???
- La prima lettere di banana non e' “a”.

Rappresentazione delle stringhe



- Le stringhe sono indicizzate e l'**indice** (= **index**) che e' l'espressione tra parentesi parte da 0.
- Una stringa lunga n ha caratteri che vanno da 0 a n-1.
- `fruit[i]` = carattere i – 1 di fruit.

```
>>> letter = fruit[0]
>>> letter2 = “banana”[0]
>>> print letter
b
>>> type(letter)
<type 'str'>
>>> print letter2
b
```

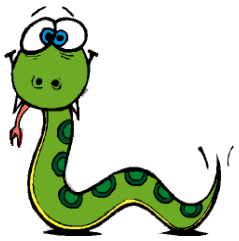


Lunghezza: **len**

```
>>> fruit = "banana"
>>> len(fruit)
6
```

- Ultima lettera:

```
length = len(fruit)
last = fruit[length]          # ERROR!
last = fruit[length-1]
```

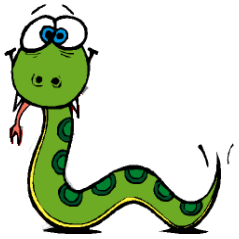


E' possibile scorrere una stringa

```
index = 0
while index < len(fruit):
  letter = fruit[index]
  print letter
  index = index + 1
```

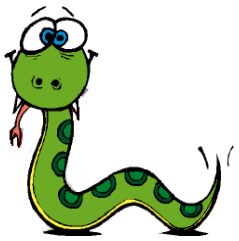
Sintassi alternativa:

```
for char in fruit:
  print char
```



Esempio

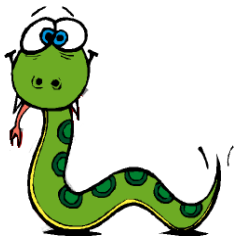
```
>>>prefixes = "JKLMNOPQ"
>>>suffix = "ack"
>>>for letter in prefixes:
...  print letter + suffix
Jack
Kack
Lack
Mack
Nack
Oack
Pack
Qack
```



Come selezionare sottostringhe

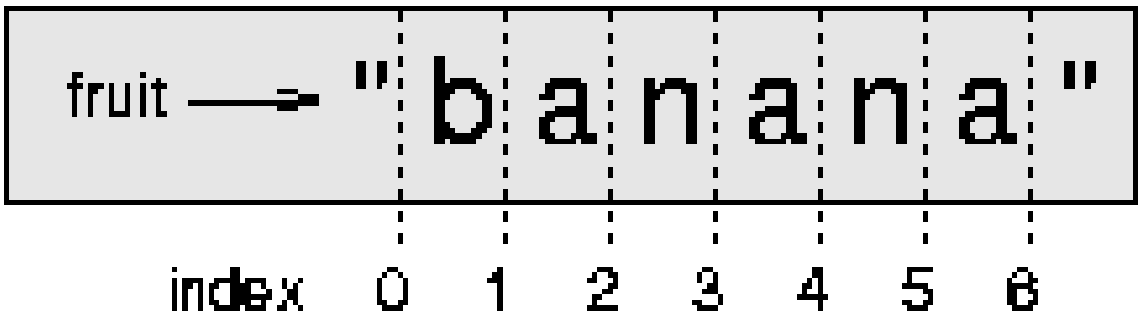
```
>>> s = "Peter, Paul, and Mary"
>>> print s[0:5]
Peter
>>> print s[7:11]
Paul
>>> print s[17:21]
Mary
```

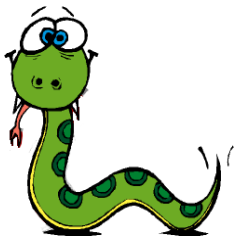
- L'operatore [n:m] restituisce la sottostringa che parte da n fino ad m escludendo l'ultimo.
- Se si omette il primo indice si parte dall'inizio.
- Se si omette l'ultimo indice si arriva fino alla fine.



Ancora sottostringhe

```
>>> fruit = "banana"
>>> fruit[:3]
'ban'
>>> fruit[3:]
'ana'
```



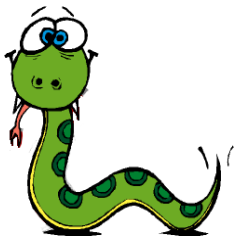


Confronto tra stringhe

```
if word == "banana":
  print  "Yes, we have no bananas!"
```

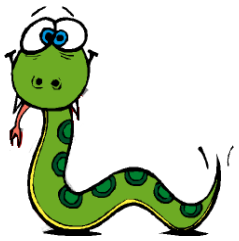
```
if word < "banana":
  print "Your word," + word + ", comes before banana."
elif word > "banana":
  print "Your word," + word + ", comes after banana."
else:
  print "Yes, we have no bananas!"
```

- Attenzione le lettere maiuscole stanno prima!
- Perche'?



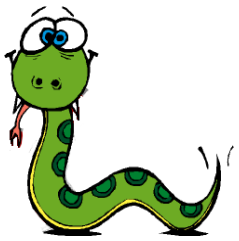
NB. le stringhe sono immutabili!

```
greeting = "Hello, world!"
greeting[0] = 'J'                 # ERROR!
print greeting
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: object doesn't support item assignment
>>> print greeting
Hello, world!
```



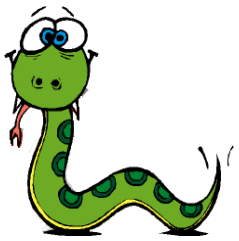
Come si cambia una stringa

```
greeting = "Hello, world!"
newGreeting = 'J' + greeting[1:]
print newGreeting
```



Cosa fa' questa funzione?

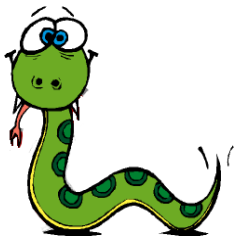
```
def find(str, ch):
  index = 0
  while index < len(str):
    if str[index] == ch:
      return index
    index = index + 1
  return -1
```



Esempio: come contare in un loop

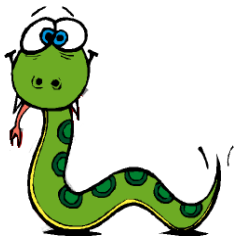
```
fruit = "banana"
count = 0
for char in fruit:
  if char == 'a':
    count = count + 1
print count
```

Il modulo **string**



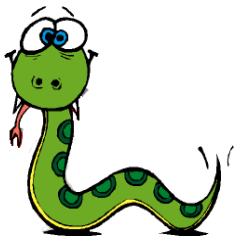
- Anche per le stringhe c'e' un modulo aggiuntivo **string** che contiene funzioni di uso generale.

```
>>> import string
>>> fruit = "banana"
>>> index = string.find(fruit, "a")
>>> print index
1
>>> string.find("banana", "na")
2
>>> string.find("banana", "na", 3)
4
>>> string.find("bob", "b", 1, 2)
-1
```



Liste in Python

- Una lista (= **list**) è un insieme di valori ordinato dove ciascun valore può essere identificato da un indice.
- I valori che appaiono in una lista sono detti i suoi elementi.
- In Python le liste sono simili alle stringhe che sono insiemi ordinati di caratteri con la differenza che gli elementi di una lista possono essere di qualsiasi tipo.
- In Python le liste, le stringhe e altri elementi che si comportano come insiemi ordinati sono detti sequenze.



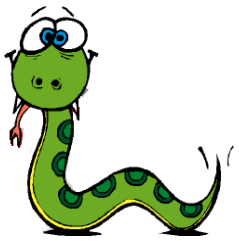
Esempi liste

- Ci sono diversi modi per creare una nuova lista, il modo più semplice è includere gli elementi tra parentesi quadre.

```
[10, 20, 30, 40]
["spam", "bungee", "swallow"]
```

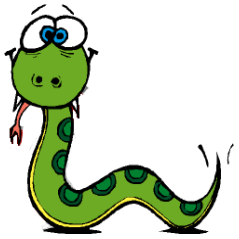
- NB. Gli elementi di una lista possono avere tipo diverso.

```
["hello", 2.0, 5, [10, 20]]
```



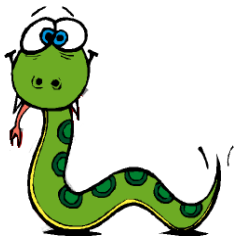
Creazione Liste di interi

```
>>> range(1,5)
[1, 2, 3, 4]
>>> range(10)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> range(1, 10, 2)
[1, 3, 5, 7, 9]
```



Esempi liste, lista vuota

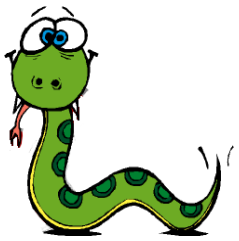
```
>>>vocabulary = ["ameliorate", "castigate", "defenestrate"]
>>>numbers = [17, 123]
>>>empty = []
>>>print vocabulary, numbers, empty
['ameliorate', 'castigate', 'defenestrate'] [17, 123] []
>>> type([1,2,3])
<type 'list'>
>>> type(vocabulary)
<type 'list'>
>>> type([])
<type 'list'>
```



Uso liste

- NB. E' possibile assegnare un valore di tipo lista ad una variabile o passarla come argomento a una funzione.
- In Python la sintassi che si utilizza per accedere agli elementi delle liste è la stessa che si usa per accedere ai caratteri nelle stringhe.
- L'espressione tra parentesi quadre specifica l'indice.

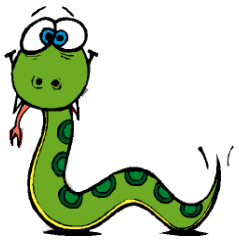
```
>>> numbers = range(1,5)
>>> print numbers
[1, 2, 3, 4]
>>> numbers[1]
2
```



Operatori su liste

- Attenzione le liste sono più potenti delle stringhe.

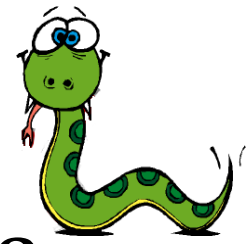
```
>>> numbers[1] = 7
>>> print numbers
[1, 7, 3, 4]
>>> numbers[3-2]  # NB questo si può fare anche con le stringhe
7
>>> numbers[1.0]  # anche qui è uguale alle stringhe
TypeError: sequence index must be integer
>>> numbers[6] = 5
IndexError: list assignment index out of range
```



Indici negativi

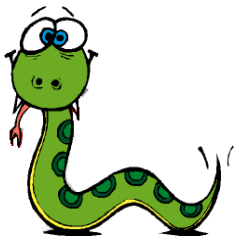
- Nelle liste se l'indice ha valore negativo conta indietro a partire dall'ultimo elemento della lista.

```
>>> numbers
[1, 7, 3, 4]
>>> numbers[-1]
4
>>> numbers[-2]
3
>>> numbers[-3]
7
>>> numbers[-4]
1
>>> numbers[-5]
IndexError: list index out of range
```



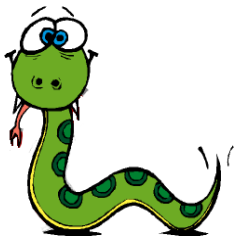
Esempio loop per scorrere una lista

```
horsemen = ["war", "famine", "pestilence", "death"]

i = 0
while i < 4:
  print horsemen[i]
  i = i + 1
```



Esempio: la funzione lunghezza

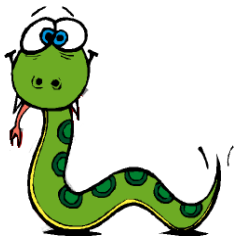
```
horsemen = ["war", "famine", "pestilence", "death"]

i = 0
while i < len(horsemen):
  print horsemen[i]
  i = i + 1

# attenzione le liste possono essere innestate ma
# una lista innestata conta come un semplice elemento

>>> len(['spam!', 1, ['Brie', 'Roquefort'], [1, 2, 3]])
4
```



L'operatore **in**

- L'operatore in è un operatore booleano che permette di verificare l'appartenenza di un elemento ad una lista.
- Si è visto in precedenza che si poteva utilizzare anche per le stringhe, in genere è utilizzabile con qualsiasi tipo di sequenza.

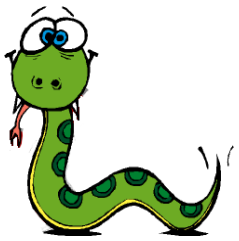
```
>>> horsemen = ['war', 'famine', 'pestilence', 'death']
>>> 'pestilence' in horsemen
1
>>> 'debauchery' in horsemen
0
>>> 'debauchery' not in horsemen
1
```



Liste e cicli

- Il comando **for** si può utilizzare anche con le liste.

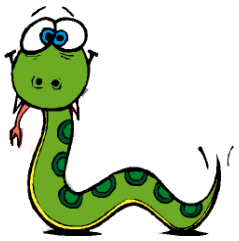
```
for VARIABLE in LIST:
  BODY

# e' equivalente a

i = 0
while i < len(LIST):
  VARIABLE = LIST[i]
  BODY
  i = i + 1
```



Esempio

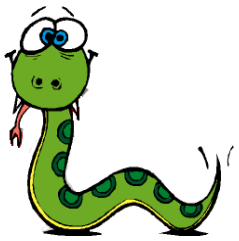
- Notare il potere espressivo di questo comando.
- E' possibile scrivere il loop visto in precedenza con solo due righe di codice.

```
for horseman in horsemen:
  print horseman
```



Esempi

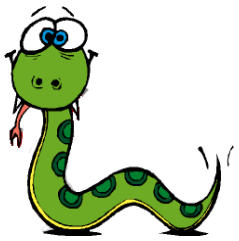
```
for number in range(20):
  if number % 2 == 0:
    print  number

for fruit in ["banana", "apple", "quince"]:
  print "I like to eat " + fruit + "s!"
```



Concatenazione e ripetizione

```
>>> a = [1, 2, 3]
>>> b = [4, 5, 6]
>>> c = a + b
>>> print c
[1, 2, 3, 4, 5, 6]
```

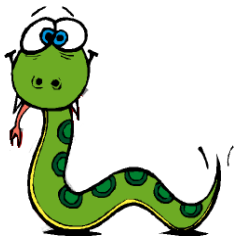
```
>>> [0] * 4
[0, 0, 0, 0]
>>> [1, 2, 3] * 3
[1, 2, 3, 1, 2, 3, 1, 2, 3]
```



Sottoliste

```
>>> list = ['a', 'b', 'c', 'd', 'e', 'f']
>>> list[1:3]
['b', 'c']
>>> list[:4]
['a', 'b', 'c', 'd']
>>> list[3:]
['d', 'e', 'f']
>>> list[:]
['a', 'b', 'c', 'd', 'e', 'f']
```

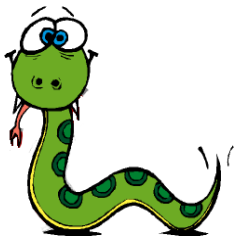
NB. è lo stesso operatore che abbiamo visto per le sottostringhe



Le liste sono modificabili

- Abbiamo già visto che è possibile modificare le liste.
- Modifica di un elemento: esempi.

```
>>> fruit = ["banana", "apple", "quince"]
>>> fruit[0] = "pear"
>>> fruit[-1] = "orange"
>>> print fruit
['pear', 'apple', 'orange']
```

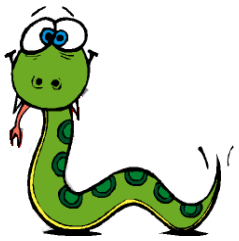


Rimuovere elementi da una lista

- Python fornisce una moda modalità alternativa più leggibile.
- **del**: elimina un elemento da una lista.

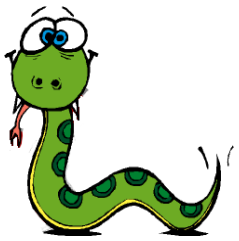
```
>>> a = ['one', 'two', 'three']
>>> del a[1]
>>> a
['one', 'three']
```

NB. del si può utilizzare anche con indici negativi.



Cancellare più elementi

```
>>> list = ['a', 'b', 'c', 'd', 'e', 'f']
>>> del list[1:5]
>>> print list
['a', 'f']
```

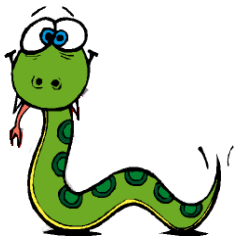


Oggetti e valori

```
a = "banana"
b = "banana"
```

- Dopo questo assegnamento sia la variabile a che b riferiscono alla stringa “banana”.
- Non siamo sicuri se effettivamente puntano alla stessa stringa.

  Due possibili stati:
  - a e b riferiscono a due cose (**oggetti** (=**objects**)) diverse che hanno lo stesso valore.
  - a e b riferiscono alla stessa cosa, cioè allo stesso oggetto.
- ***Cosa succede in Python?***

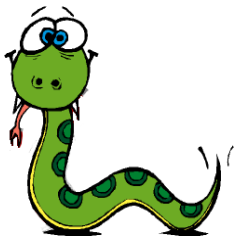


Oggetti e identificatori

- In Python ogni oggetto ha un identificatore unico, questo identificatore si può ottenere con la funzione **id**.
- Per verificare se è stato creato lo stesso oggetto bisogna controllare gli id associati alle variabili.

```
>>> id(a)
135044008
>>> id(b)
135044008
```

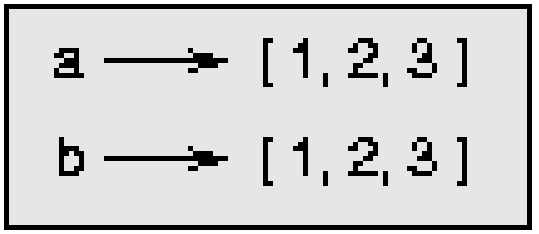
NB. prima viene valutata la variabile quindi l'id è quello dell'oggetto non quello della variabile.



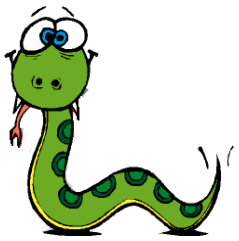
E con le liste?

- Le liste si comportano in modo diverso! Ovvero vengono creati due oggetti distinti.

```
>>> a = [1, 2, 3]
>>> b = [1, 2, 3]
>>> id(a)
135045528
>>> id(b)
135041704
>>> a == b
True
```



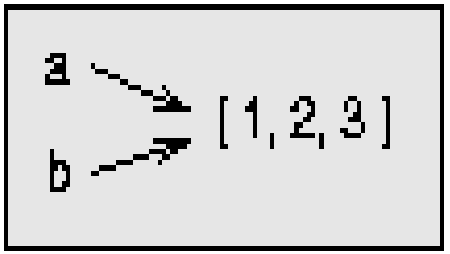
NB. a e b hanno lo stesso valore ma non riferiscono allo stesso oggetto.
NB2. L'ugualianza funziona lo stesso!



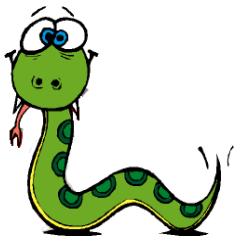
Aliasing

- Dato che le variabili si riferiscono ad oggetti, se noi assegnamo una variabile ad un altra, le due variabili alla fine riferiscono (puntano) allo stesso oggetto.

```
>>> a = [1, 2, 3]
>>> b = a
```



Aliasing: la stessa lista ha due nomi different.



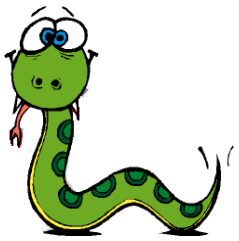
Problemi dell'aliasing

- I cambiamenti fatti a una variabile influiscono anche sull'altra.

```
>>> b[0] = 5
>>> print a
[5, 2, 3]
```

- Questo comportamento a volte può essere utile ma in tanti altri casi l'aliasing può provocare delle sorprese.
- Si consiglia di evitare l'aliasing quando si ha che fare con oggetti che vengono modificati.
- Con oggetti che non cambiano invece (come le stringhe) non ci sono problemi.

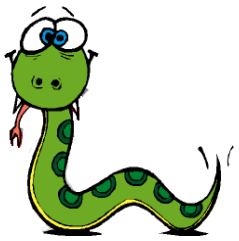
Clonazione di liste



- Se si vuole modificare una lista lasciando invariata l'originale è necessario fare una copia.
- Quello che si faceva prima era invece assegnare a una variabile lo stesso valore.
- Il modo più semplice per fare una copia è utilizzare l'operatore sottolista (slice).

```
>>> a = [1, 2, 3]
>>> b = a[:]
>>> print b
[1, 2, 3]
```

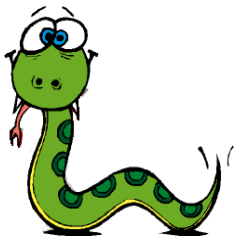
In questo modo si assegna a b una copia completa della lista a.



Cloning

- In genere una qualsiasi sottolista creata con l'operatore (slice) è copiata dalla lista originale.
- Ora è possibile cambiare b senza influire su a.

```
>>> b[0] = 5
>>> print a
[1, 2, 3]
```

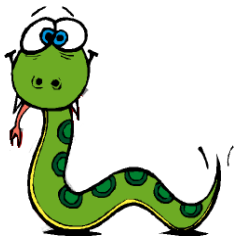


Liste passate come parametri

- NB. Quando una lista viene passata come parametro in realtà si passa il riferimento non una copia della lista.

```
def head(list):
  return list[0]


# si usa cosí

>>> numbers = [1, 2, 3]
>>> head(numbers)
1
```

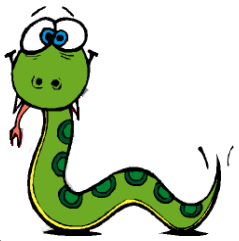
Il parametro list e la variabile (argomento) numbers sono alias per la stessa lista.



Liste e parametri

- Se una funzione modifica una lista passata come parametro, si modifica anche la lista originale.

```
def deleteHead(list):
  del list[0]

>>> numbers = [1, 2, 3]
>>> deleteHead(numbers)
>>> print numbers
[2, 3]
```

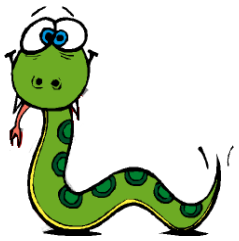


Restituzione di liste come risultato

- Quando una funzione restituisce una lista, restituisce un riferimento a quella lista.

```
def tail(list):
  return list[1:]

>>> numbers = [1, 2, 3]
>>> rest = tail(numbers)
>>> print rest
[2, 3]
```

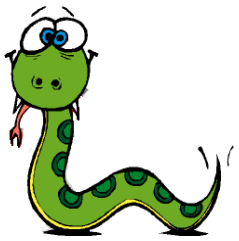
NB. dato che il risultato è ottenuto con l'operatore sottolista si tratta di una nuova lista.



Le tuple

- Due strutture dati composte:
  - Stringhe: non modificabili (immutabili)
  - Liste: modificabili (mutabili)
- Le **tuple** sono strutture dati simili alle liste ma immutabili.
- Una tupla è una lista di valori separati da una virgola.

```
>>> tuple = 'a', 'b', 'c', 'd', 'e'
```



Esempi di tuple

- Anche se non è strettamente necessario, si consiglia di includere le tuple tra parentesi.

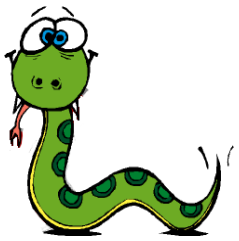
```
>>> tuple = ('a', 'b', 'c', 'd', 'e')
```

- NB. per creare una tupla con un solo elemento è necessario includere la virgola finale:

```
>>> t1 = ('a',)
>>> type(t1)
<type 'tuple'>
```

```
>>> t2 = ('a')
>>> type(t2)
<type 'string'>
```

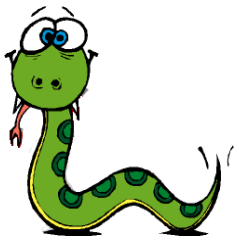
Si tratta della stringa “a” tra parentesi



Operazioni sulle tuple

- A parte le differenze sintattiche le operazioni sulle tuple sono le stesse che sulle liste, tranne che ....
- Ad esempio la selezione con indice di un elemento e operatore slice.

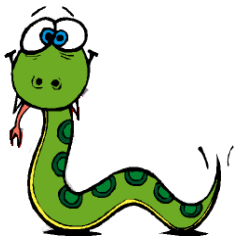
```
>>> tuple = ('a', 'b', 'c', 'd', 'e')
>>> tuple[0]
'a'
>>> tuple[1:3]
('b', 'c')
>>> tuple[0] = 'A'
TypeError: object doesn't support item assignment
```



Operazioni sulle tuple

- Anche se non si può modificare una tupla è sempre possibile sostituirla con un altra tupla.

```
>>> tuple = ('A',) + tuple[1:]
>>> tuple
('A', 'b', 'c', 'd', 'e')
```



Assegnamenti a tuple

- La tipica tecnica per scambiare i valori di di due variabili utilizza una variabile temporanea:

```
>>> temp = a
>>> a = b
>>> b = temp
```

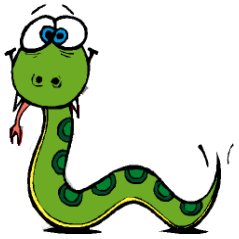
- Questo scambio se fatto più volte può risultare ingombrante in un programma.
- Python fornisce una versione elegante basata sulle tuple:

```
>>> a, b = b, a
```

NB. attenzione alla semantica!

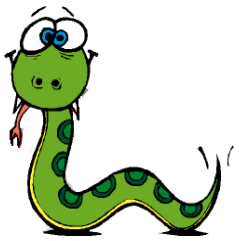
Semantica assegnamento con tuple

- La parte sinistra una tupla di variabili; la parte destra una tupla di valori.
- Ogni valore viene assegnato alla rispettiva variabile.
- Prima di fare un qualsiasi assegnamento vengono valutate tutte le espressioni della parte destra.
- Il numero di variabili sulla sinistra deve essere uguale al numero di valori sulla destra.



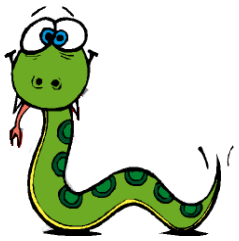
Esempi

```
>>> a, b, c, d = 1, 2, 3
ValueError: unpack tuple of wrong size
```



Dizionari

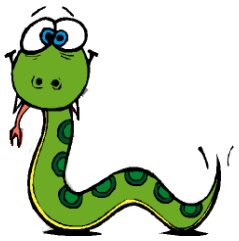
- I tipi di dati strutturati che sono stati presentati fino ad ora (stringhe, liste e tuple) utilizzavano indici interi per accedere alle diverse parti delle strutture dati.
- I **dizionari** (= **dictionary**) sono simili ai tipi di dati strutturati che abbiamo visto fino ad ora con la differenza che possono utilizzare un qualsiasi tipo immutabile come indice.
- Ad esempio è possibile creare un dizionario che permette di tradurre parole inglese in spagnolo; per tale dizionario gli indici sono stringhe.



Esempio

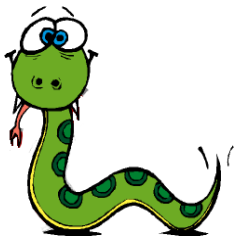
- Si può iniziare creando un dizionario vuoto e poi riempendolo.
- Il dizionario vuoto si indica con {}.

```
>>> eng2sp = {}
>>> eng2sp['one'] = 'uno'
>>> eng2sp['two'] = 'dos'
>>> print eng2sp
{'one': 'uno', 'two': 'dos'}
```



Osservazioni

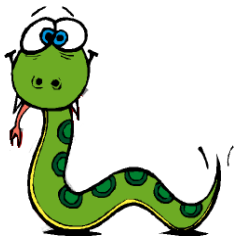
- Gli elementi di un dizionario appaiono in una lista separati da virgole.
- Ciascun elemento contiene un indice e un valore separati da “:”.
- Per questo motivo gli elementi sono definiti **coppie chiave-valore** (=**key-value pairs**)



Ancora un esempio

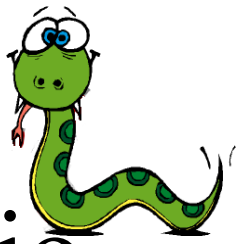
```
>>> eng2sp = {'one': 'uno', 'two': 'dos', 'three': 'tres'}
>>> print eng2sp
{'one': 'uno', 'three': 'tres', 'two': 'dos'}
```

Che è successo?



Osservazioni

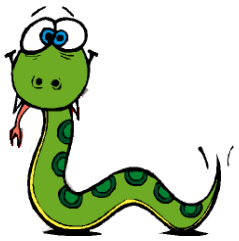
- Le coppie chiave valore non sono ordinate!
- Fortunatamente c'è una buona ragione che permette di fare questo: non è possibile accedere agli elementi di un dizionario tramite un indice intero.
- Solo le chiavi che sono state inserite contano.
- NB. le chiavi possono essere interi, ma non c'è un indice che permette di scorrere un dizionario come una lista.



Accesso agli elementi di un dizionario

- È possibile utilizzare una chiave per accedere agli elementi di un dizionario.
- NB. la chiave permette di accedere all'elemento senza il bisogno di scorrere il dizionario.

```
>>> print eng2sp['two']
'dos'
```



Operazioni su dizionari

```
>>> inventory = {'apples': 430, 'bananas': 312, 'oranges': 525,
'pears': 217}
>>> print inventory
{'oranges': 525, 'apples': 430, 'pears': 217, 'bananas': 312}
>>> del inventory['pears']
>>> print inventory
{'oranges': 525, 'apples': 430, 'bananas': 312}
# OPPURE
>>> inventory['pears'] = 0
>>> print inventory
{'oranges': 525, 'apples': 430, 'pears': 0, 'bananas': 312}
>>> len(inventory)
4
```

NB. I dizionari sono mutabili!