

# Intelligenza Artificiale

*Anno Accademico 2022 - 2023*

***Introduzione al  
Linguaggio Python***

### **Norme di utilizzo dei materiali didattici**

La visione e l'utilizzo del presente materiale didattico è riservato agli utenti iscritti al corso al solo fine di studio e approfondimento didattico.

L'utente che accede alla sezione e-learning e che ne consulta i contenuti è tenuto a rispettare le disposizioni legislative che tutelano il diritto d'autore e pertanto è fatto espresso divieto di riprodurre, pubblicare o distribuire i materiali tratti dal presente sito, anche in forma parziale, fatta salva la possibilità di realizzare un'unica copia del materiale, in formato cartaceo e/o digitale, all'esclusivo fine di studio.

L'utente è responsabile per l'uso improprio o non autorizzato del materiale pubblicato effettuato in violazione delle suddette disposizioni.

# SOMMARIO

- Introduzione al Linguaggio
- Variabili, Espressioni e Istruzioni
- Esecuzione Condizionale
- Funzioni in Python
- Iterazione
- Stringhe

# SOMMARIO

## Strutture Dati di base

- Liste
- Dizionari
- Tuple
- Set

# INTRODUZIONE AL LINGUAGGIO

- Python è un potente linguaggio di programmazione general-purpose ideato da Guido van Rossum nel 1989.
- Esso è classificato come un linguaggio ad alto livello poiché gestisce automaticamente le più fondamentali operazioni (i.e., gestione della memoria) effettuate al livello del processore (“codice macchina”).
- E' considerato più ad alto livello ad esempio del linguaggio C, a causa della sua espressiva sintassi (che a volte è molto vicina al linguaggio naturale) e della ricca varietà di strutture dati native come le liste, le tuple, gli insiemi e i dizionari.

# INTRODUZIONE AL LINGUAGGIO

Python è un linguaggio di programmazione:

- Interpretato
- Di alto livello
- Semplice da imparare e usare
- Potente e produttivo
- Ottimo anche come primo linguaggio
- Estensibile
- Con tipizzazione forte e dinamica
- Open source ([www.python.org](http://www.python.org)) e multiplatforma
- Materiale didattico disponibile (e.g., <https://docs.python.org/3/tutorial/>)

# VARIABILI, ESPRESSIONI E ISTRUZIONI

## VALORI E TIPI

Usiamo il comando **python** per avviare l'interprete:

```
$ python
```

```
>>> print(4)
```

```
4
```

Per conoscere il tipo di un valore:

```
>>> type('Hello World!')
```

```
<class 'str'>
```

```
>>> type(17)
```

```
<class 'int'>
```

```
>>> type(3.2)
```

```
<class 'float'>
```

# VARIABILI, ESPRESSIONI E ISTRUZIONI

## VALORI E TIPI

Vediamo ora il tipo di valori come "17" e "3.2":

```
>>> type('17')
```

```
<class 'str'>
```

```
>>> type('3.2')
```

```
<class 'str'>
```

Tali valori sono delle stringhe.



# VARIABILI, ESPRESSIONI E ISTRUZIONI

## VARIABILI

Una istruzione di assegnazione crea nuove variabili e gli assegna dei valori:

```
>>> messaggio = 'Tanti auguri!'
```

```
>>> n = 25
```

```
>>> pi = 3.141592
```

Per visualizzare il valore di una variabile possiamo usare l'istruzione **print**:

```
>>> print(n)
```

```
25
```

```
>>> print(pi)
```

```
3.141592
```

# VARIABILI, ESPRESSIONI E ISTRUZIONI

## VARIABILI

Il tipo di una variabile è il tipo del valore ad essa associato:

```
>>> type(messaggio)
```

```
<class 'str'>
```

```
>>> type(n)
```

```
<class 'int'>
```

```
>>> type(pi)
```

```
<class 'float'>
```

# VARIABILI, ESPRESSIONI E ISTRUZIONI

## NOMI DELLE VARIABILI E KEYWORDS

- I nomi delle variabili possono avere una lunghezza arbitraria.
- Possono contenere sia lettere che numeri, ma non possono iniziare con un numero. E' legale utilizzare lettere maiuscole, ma è consigliabile utilizzare una minuscola come primo carattere del nome.
- Python ha 33 parole chiave (keywords), che ovviamente non possono essere usate come nomi di variabili:

<b>and</b>	<b>del</b>	<b>from</b>	<b>None</b>	<b>True</b>
<b>as</b>	<b>elif</b>	<b>global</b>	<b>nonlocal</b>	<b>try</b>
<b>assert</b>	<b>else</b>	<b>if</b>	<b>not</b>	<b>while</b>
<b>break</b>	<b>except</b>	<b>import</b>	<b>or</b>	<b>with</b>
<b>Class</b>	<b>False</b>	<b>in</b>	<b>pass</b>	<b>yield</b>
<b>continue</b>	<b>finally</b>	<b>is</b>	<b>raise</b>	
<b>def</b>	<b>for</b>	<b>lambda</b>	<b>return</b>	

# VARIABILI, ESPRESSIONI E ISTRUZIONI

## OPERATORI E OPERANDI

- Gli **operatori** sono simboli speciali che consentono di effettuare computazioni come l'addizione e la moltiplicazione.
- I valori a cui si applicano gli operatori sono detti **operandi**.
- Gli operatori `+`, `-`, `*`, `/` e `**` sono relativi a: addizione, sottrazione, moltiplicazione, divisione ed elevamento a potenza.

Esempio:

```
>>> minuti = 59
```

```
>>> minuti/60
```

```
0.9833333333333333
```

# VARIABILI, ESPRESSIONI E ISTRUZIONI

## OPERATORI E OPERANDI

In Python 2 otterremmo invece questo risultato:

```
>>> minuti = 59
```

```
>>> minuti/60
```

```
0
```

Per avere lo stesso risultato in Python 3 possiamo utilizzare l'operatore `//` (**flored division**):

```
>>> minuti = 59
```

```
>>> minuti//60
```

```
0
```

# VARIABILI, ESPRESSIONI E ISTRUZIONI

## ESPRESSIONI

- Una **espressione** è una combinazione di valori, variabili e operatori.
- Un valore è di per sé una espressione, così come una variabile.

Esempi:

**17**

**x**

**x + 17**

- Se si inserisce una espressione in modalità interattiva, l'interprete la **valuta** e mostra il risultato:

```
>>> 1 + 1
```

```
2
```

# VARIABILI, ESPRESSIONI E ISTRUZIONI

## ORDINE DELLE OPERAZIONI

- Quando in una espressione compaiono più operatori, l'ordine di valutazione dipende dalle seguenti **regole di precedenza**:
  1. Le parentesi hanno la massima precedenza.
  2. Elevamento a potenza.
  3. Moltiplicazione e la divisione, che hanno la stessa priorità.
  4. Addizione e sottrazione, che hanno la stessa priorità.
  5. Gli operatori con lo stesso ordine di priorità vengono valutati da sinistra a destra.

# VARIABILI, ESPRESSIONI E ISTRUZIONI

## L'OPERATORE MODULO

- L'operatore **modulo** lavora sui numeri interi e restituisce il resto della divisione quando il primo operando è diviso per il secondo.
- In Python l'operatore modulo è il simbolo percento: %

Esempio:

```
>>> quoziente = 7 // 3
```

```
>>> print(quoziente)
```

```
2
```

```
>>> resto = 7 % 3
```

```
>>> print(resto)
```

```
1
```




# VARIABILI, ESPRESSIONI E ISTRUZIONI

## INSERIMENTO DATI DA INPUT

- Python dispone di una funzione *built-in* chiamata **input** che permette di acquisire un input da tastiera (in Python 2 tale funzione è **raw\_input**).

Esempio:

```
>>> dato = input()
Frase inserita
>>> print(dato)
Frase inserita
>>> nome = input('Qual è il tuo nome?\n')
Qual è il tuo nome?
Mario
>>> print(nome)
Mario
```



newline

# VARIABILI, ESPRESSIONI E ISTRUZIONI

## INSERIMENTO DATI DA INPUT

- Se l'utente deve inserire un intero, si può convertire il valore inserito in **int** mediante la funzione **int()**.

Esempio:

```
>>> prompt = 'Inserire la velocità\n'
>>> speed = input(prompt)
Inserire la velocità
50
>>> int(speed)
50
>>> int(speed) + 10
60
```

# VARIABILI, ESPRESSIONI E ISTRUZIONI

## COMMENTI

- In Python possono essere inseriti commenti preceduti dal simbolo #.

Esempio:

```
# compute the percentage of the hour that has elapsed
```

```
Percentage = (minute * 100) / 60
```

oppure:

```
Percentage = (minute * 100) / 60    # percentage of an hour
```

# ESECUZIONE CONDIZIONALE

## ESPRESSIONI BOOLEANE

- Una *espressione booleana* è una espressione che è o vera o falsa.

Nei seguenti esempi usiamo l'operatore `==` che confronta due operandi:

```
>>> 5 == 5
```

```
True
```

```
>>> 5 == 6
```

```
False
```

```
>>> type(True)
```

```
<class 'bool'>
```

```
>>> type(False)
```

```
<class 'bool'>
```

# ESECUZIONE CONDIZIONALE

## ESPRESSIONI BOOLEANE

● Ecco la lista dei *comparison operators* disponibili in Python:

```
x == y          # x è uguale a y
x != y          # x non è uguale a y
x > y           # x è maggiore di y
x < y           # x è minore di y
x >= y          # x è maggiore o uguale a y
x <= y          # x è minore o uguale a y
x is y          # x è lo stesso di y
x is not y      # x non è lo stesso di y
```

# ESECUZIONE CONDIZIONALE

## OPERATORI LOGICI

● Gli *operatori logici* sono: **and**, **or** e **not**.

Esempio:

```
x > 0 and x < 10
```

è vera solo se **x** è maggiore di **0** e minore di **10**.

```
n%2 == 0 or n%3 == 0
```

è vera se almeno una delle due condizioni è vera, ossia se **n** è divisibile per 2 o per 3.

```
not(x > y) è vera se x > y è falsa.
```

Qualsiasi numero diverso da zero è interpretato con “vero”:

```
>>> 17 and True
```

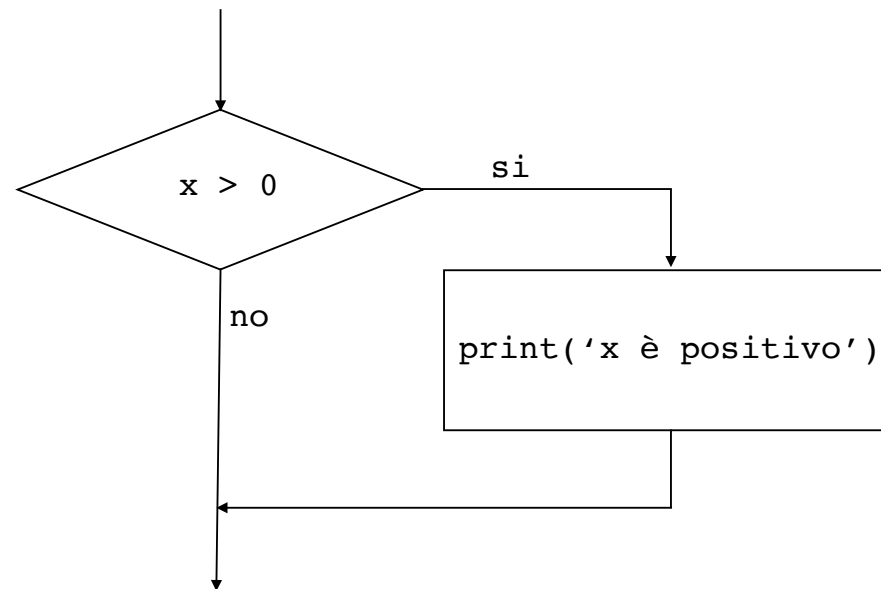
```
True
```

# ESECUZIONE CONDIZIONALE

## ISTRUZIONE “IF”

Esempio:

```
if x > 0 :  
    print('x è positivo')
```

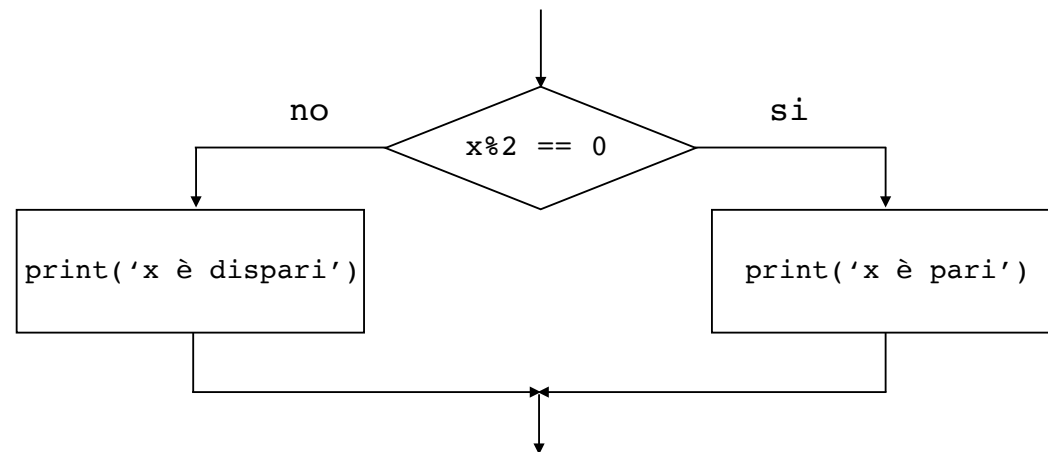


# ESECUZIONE CONDIZIONALE

## ISTRUZIONE “IF-ELSE”

Esempio:

```
if x%2 == 0 :  
    print('x è pari')  
else :  
    print('x è dispari')
```



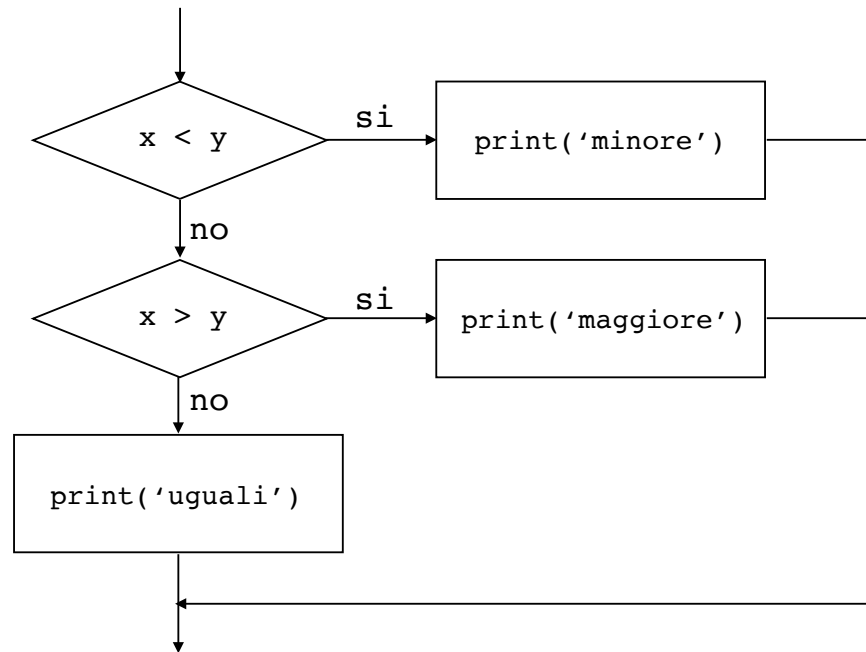


# ESECUZIONE CONDIZIONALE

## “IF” CONCATENATI

Esempio:

```
if x < y:  
    print('x è minore di y')  
elif x > y:  
    print('x è maggiore di y')  
else:  
    print('x e y sono uguali')
```

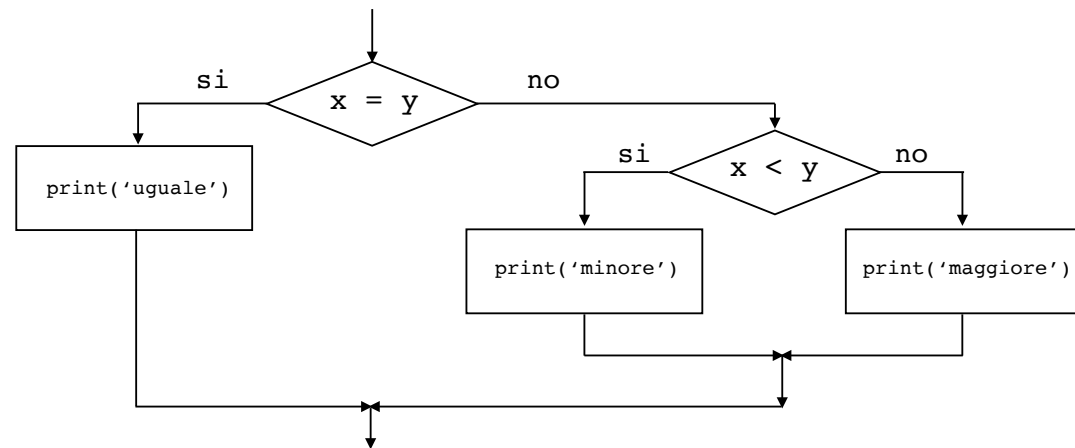


# ESECUZIONE CONDIZIONALE

## “IF” ANNIDATI

Esempio:

```
if x == y:
    print('x e y sono uguali')
else:
    if x < y:
        print('x è minore di y')
    else:
        print('x è maggiore di y')
```



# ESECUZIONE CONDIZIONALE

## GESTIONE ECCEZIONI CON TRY E EXCEPT

- E' possibile gestire le eccezioni che si verificano ad esempio quando si introduce in input un dato di tipo errato.

Esempio:

Supponiamo di utilizzare il seguente programma (**fahren.py**) per convertire la temperatura da Fahrenheit a Celsius:

```
inp = input('Enter Fahrenheit Temperature: ')
fahr = float(inp)
cel = (fahr - 32.0) * 5.0/9.0
print(cel)
```

# ESECUZIONE CONDIZIONALE

## GESTIONE ECCEZIONI CON TRY E EXCEPT

Vediamo due esempi di esecuzione:

```
python fahren.py
```

```
Enter Fahrenheit Temperature: 72
```

```
22.22222222222222
```

```
python fahren.py
```

```
Enter Fahrenheit Temperature: pippo
```

```
Traceback (most recent call last):
```

```
  File "fahren.py", line 2, in <module>
```

```
    fahr = float(inp)
```

```
ValueError: could not convert string to float: 'pippo'
```

# ESECUZIONE CONDIZIONALE

## GESTIONE ECCEZIONI CON TRY E EXCEPT

Possiamo riscrivere il programma come segue:

```
inp = input('Enter Fahrenheit Temperature: ')
try:
    fahr = float(inp)
    cel = (fahr - 32.0) * 5.0/9.0
    print(cel)
except:
    print('Please enter a number')
```

# ESECUZIONE CONDIZIONALE

## GESTIONE ECCEZIONI CON TRY E EXCEPT

- Python comincia eseguendo le istruzioni del blocco **try**. Se tutto va bene, salta il blocco **except** e procede con le istruzioni successive.
- Se si verifica una eccezione nel blocco **try**, Python salta il blocco ed esegue la sequenza di istruzioni presenti nel blocco **except**.

Esempio:

```
Enter Fahrenheit Temperature: 72
```

```
22.22222222222222
```

```
Enter Fahrenheit Temperature: pippo
```

```
Please enter a number
```

# FUNZIONI IN PYTHON

## CHIAMATA DI FUNZIONI

- Una funzione è una sequenza specifica di istruzioni a cui è stato attribuito un nome, che ha lo scopo di eseguire un'operazione o calcolo particolare. Una volta definita, è possibile “chiamare” una funzione per mezzo del suo nome.
- Abbiamo già visto in precedenza un esempio di *chiamata di funzione*:

```
>>> type(32)  
<class 'int'>
```

- In questo esempio il nome della funzione è **type** mentre il numero **32** tra parentesi è l'argomento della funzione. L'argomento può essere un valore o una variabile che stiamo passando alla funzione come input.

# FUNZIONI IN PYTHON

## FUNZIONI “BUILT-IN”

- In Python sono integrate un gran numero di funzioni utili, create per svolgere le attività più comuni, che possiamo usare senza doverle definire.
- Ad esempio le funzioni **max** e **min** ci forniscono rispettivamente i valori più grandi e più piccoli di una stringa:

```
>>> max('Hello world')  
'w'  
>>> min('Hello world')  
' '
```

- La funzione **max** ci indica che il “carattere più grande” nella stringa è la lettera “w” mentre **min** ci mostra che il carattere più piccolo è lo spazio.



# FUNZIONI IN PYTHON

## FUNZIONI “BUILT-IN”

- Un'altra funzione molto utilizzata è la funzione **len** che ci dice quanti item ci sono nell'argomento analizzato.
- Nell'esempio seguente l'argomento di **len** è una stringa e perciò viene restituito il numero di caratteri nella stringa:

```
>>> len('Hello world')  
11
```

- Come vedremo successivamente, tali funzioni non sono progettate per operare solo con le stringhe ma possono essere applicate su un qualsiasi insieme di valori.

# FUNZIONI IN PYTHON

## CONVERSIONE DEI TIPI DI DATO

- Python fornisce anche funzioni built-in che permettono di convertire i valori da un tipo ad un altro.
- Ad esempio la funzione **int** accetta qualsiasi valore e, qualora sia possibile, lo converte in un numero intero. In caso contrario segnala un messaggio di errore:

```
>>> int('32')
```

```
32
```

```
>>> int('Hello')
```

```
ValueError: invalid literal for int() with base 10: 'Hello'
```

# FUNZIONI IN PYTHON

## CONVERSIONE DEI TIPI DI DATO

- La funzione **int** può convertire valori in virgola mobile in numeri interi eliminando semplicemente la parte frazionaria senza arrotondare il dato:

```
>>> int(3.99999)
```

```
3
```

```
>>> int(-2.3)
```

```
-2
```

- La funzione **float** converte numeri interi e stringhe in numeri in virgola mobile:

```
>>> float(32)
```

```
32.0
```

```
>>> float('3.14159')
```

```
3.14159
```

# FUNZIONI IN PYTHON

## CONVERSIONE DEI TIPI DI DATO

- Infine, la funzione **str** converte il suo argomento in una stringa:

```
>>> str(32)
```

```
'32'
```

```
>>> str(3.14159)
```

```
'3.14159'
```

# FUNZIONI IN PYTHON

## FUNZIONI MATEMATICHE

- Il modulo **math** di Python consente di eseguire la maggior parte delle più comuni funzioni matematiche. Essendo un modulo, va importato prima di poterlo utilizzare:

```
>>> import math
```

- Questa istruzione crea un *object module* chiamato math.

# FUNZIONI IN PYTHON

## FUNZIONI MATEMATICHE

- Per accedere a una delle funzioni del modulo **math** è necessario specificare il nome del modulo e della funzione desiderata separati da un punto. Questo particolare formato viene chiamato *dot notation* (notazione punto). Ad es.:

```
>>> ratio = signal_power / noise_power
```

```
>>> decibels = 10 * math.log10(ratio)
```

```
>>> radians = 0.7
```

```
>>> height = math.sin(radians)
```

# FUNZIONI IN PYTHON

## FUNZIONI MATEMATICHE

- Le funzioni trigonometriche accettano un argomento in radianti. Per convertire i gradi in radianti occorre dividere il dato in ingresso per 360 e moltiplicarlo per  $2\pi$ :

```
>>> degrees = 45
>>> radians = degrees / 360.0 * 2 * math.pi
>>> math.sin(radians)
0.7071067811865476
```

- L'espressione **math.pi** richiama la variabile **pi** dal modulo math il cui valore è un'approssimazione di  $\pi$  a circa 15 cifre.

# FUNZIONI IN PYTHON

## NUMERI CASUALI

- Dato il medesimo input, la maggior parte dei programmi per computer genera sempre lo stesso output. Per tale motivo vengono detti *deterministici*. Il determinismo è di solito una cosa auspicabile poiché ci aspettiamo che lo stesso calcolo produca sempre il medesimo risultato.
- Ci sono casi in cui, tuttavia, desideriamo che il computer sia imprevedibile.



# FUNZIONI IN PYTHON

## NUMERI CASUALI

- Realizzare un programma veramente non deterministico non è così semplice come potrebbe sembrare a prima vista, anche se esistono modi per farlo almeno sembrare tale.
- Uno di questi è usare algoritmi che generano numeri *pseudo-casuali*.
- I numeri pseudo-casuali non sono veramente casuali perché sono generati da un calcolo deterministico, ma osservando solo i numeri è quasi impossibile distinguerli da quelli casuali.

# FUNZIONI IN PYTHON

## NUMERI CASUALI

- Il modulo **random** fornisce funzioni che generano numeri pseudo-casuali (che chiameremo semplicemente “casuali” da qui in poi).
- La funzione **random** restituisce un numero decimale casuale compreso tra 0.0 e 1.0 (comprendendo 0.0 ma non 1.0). Ogni volta che si chiama **random** si ottiene il numero successivo di una lunga serie. Ad es.:

```
import random

for i in range(10):
    x = random.random()
    print(x)
```

# FUNZIONI IN PYTHON

## NUMERI CASUALI

- Il ciclo **for** precedente ci fornirà una lista di 10 numeri casuali compresi tra 0.0 e 1.0 (1.0 escluso), come ad esempio i seguenti:

```
0.11873666388894799  
0.2089715608073831  
0.3526066972736147  
0.5864002586446272  
0.16376975732542987  
0.13871794823211492  
0.5761320986066906  
0.29320107107181603  
0.9559592563967794  
0.02126065574782532
```

# FUNZIONI IN PYTHON

## NUMERI CASUALI

- La funzione **random** è solo una delle molte funzioni che gestiscono i numeri casuali.
- La funzione **randint** accetta i parametri low e high e restituisce un intero compreso tra i due estremi (estremi inclusi). Ad es.:

```
>>> random.randint(5, 10)
```

```
8
```

```
>>> random.randint(5, 10)
```

```
5
```

# FUNZIONI IN PYTHON

## NUMERI CASUALI

- Tramite **choice** è possibile scegliere casualmente un elemento da una sequenza:

```
>>> t = [1, 2, 3]
>>> random.choice(t)
2
>>> random.choice(t)
1
```

- Il modulo **random** fornisce anche funzioni per generare valori casuali da distribuzioni continue come la gaussiana, l'esponenziale, ecc.

# FUNZIONI IN PYTHON

## DEFINIZIONE DI NUOVE FUNZIONI

- Mediante la **definizione di funzione** è possibile specificare il nome di una nuova funzione composta da una sequenza di istruzioni. Una volta definita, è possibile richiamarla e riutilizzarla a piacimento all'interno del nostro programma.

Esempio:

```
def print_lyrics():  
    print("I'm a lumberjack, and I'm okay.")  
    print("I sleep all night and I work all day.")
```

- **def** è la parola chiave che indica che si tratta di una definizione di funzione. Il nome di questa funzione è **print\_lyrics**. Essa è una funzione senza argomenti.

# FUNZIONI IN PYTHON

## DEFINIZIONE DI NUOVE FUNZIONI

- Una funzione è composta da un *header*, la prima riga della funzione, ed un *corpo* composto dal resto delle istruzioni. L'header deve sempre terminare con due punti e il corpo, che deve essere indentato per convenzione di quattro spazi, può contenere un numero qualsiasi di istruzioni.
- Se si scrive una definizione di funzione in modalità interattiva, l'interprete continuerà a visualizzare tre punti di sospensione (...) fino a quando non si inserisce una riga vuota per indicare la fine del corpo:

```
>>> def print_lyrics():  
...     print("I'm a lumberjack, and I'm okay.")  
...     print("I sleep all night and I work all day.")  
...
```

# FUNZIONI IN PYTHON

## DEFINIZIONE DI NUOVE FUNZIONI

- La definizione di una funzione crea una variabile con lo stesso nome:

```
>>> print(print_lyrics)
<function print_lyrics at 0xb7e99e9c>
>>> print(type(print_lyrics))
<class 'function'>
```

- Il valore di **print\_lyrics** è un *oggetto funzione* di tipo “funzione”.
- La sintassi per chiamare la nuova funzione è la stessa di quella utilizzata per le funzioni built-in:

```
>>> print_lyrics()
I'm a lumberjack, and I'm okay.
I sleep all night and I work all day.
```



# FUNZIONI IN PYTHON

## DEFINIZIONE DI NUOVE FUNZIONI

- Una volta definita una funzione, è possibile richiamarla all'interno di un'altra funzione. Ad esempio:

```
def repeat_lyrics():  
    print_lyrics()  
    print_lyrics()
```

- Chiamando la funzione **repeat\_lyrics** otterremo questo risultato:

```
>>> repeat_lyrics()  
  
I'm a lumberjack, and I'm okay.  
I sleep all night and I work all day.  
I'm a lumberjack, and I'm okay.  
I sleep all night and I work all day.
```

# FUNZIONI IN PYTHON

## DEFINIZIONI E USI

- Se mettiamo insieme i vari frammenti di codice visti in precedenza, otteniamo il seguente programma:

```
def print_lyrics():  
    print("I'm a lumberjack, and I'm okay.")  
    print("I sleep all night and I work all day.")  
  
def repeat_lyrics():  
    print_lyrics()  
    print_lyrics()  
  
repeat_lyrics()
```

# FUNZIONI IN PYTHON

## PARAMETRI E ARGOMENTI

- Alcune funzioni built-in viste in precedenza richiedono degli argomenti. Ad esempio la funzione **math.sin** ha bisogno di un numero come argomento.
- Altre funzioni richiedono più di un argomento: **math.pow** ne richiede due, la base e l'esponente.
- All'interno della funzione, gli argomenti sono assegnati a variabili chiamate *parametri*.

# FUNZIONI IN PYTHON

## PARAMETRI E ARGOMENTI

- Ecco un esempio di una funzione definita dall'utente che riceve un argomento:

```
def print_twice(bruce):  
    print(bruce)  
    print(bruce)
```

- Questa funzione assegna l'argomento ad un parametro chiamato **bruce**. Quando viene chiamata la funzione, questa stampa due volte il valore del parametro (qualunque esso sia).

# FUNZIONI IN PYTHON

## PARAMETRI E ARGOMENTI

- Tale funzione opera con qualsiasi valore che possa essere visualizzato:

```
>>> print_twice('Spam'):
```

```
Spam
```

```
Spam
```

```
>>> print_twice(17):
```

```
17
```

```
17
```

```
>>> import math
```

```
>>> print_twice(math.pi)
```

```
3.141592653589793
```

```
3.141592653589793
```

# FUNZIONI IN PYTHON

## PARAMETRI E ARGOMENTI

- Le stesse regole di composizione che si applicano alle funzioni built-in si applicano anche a quelle definite dall'utente. Pertanto possiamo usare qualsiasi tipo di espressione come argomento per **print\_twice**:

```
>>> print_twice('Spam ' * 4):
```

```
Spam Spam Spam Spam
```

```
Spam Spam Spam Spam
```

```
>>> print_twice(math.cos(math.pi))
```

```
-1.0
```

```
-1.0
```

# FUNZIONI IN PYTHON

## PARAMETRI E ARGOMENTI

- E' possibile utilizzare inoltre una variabile come argomento:

```
>>> michael = 'Eric, the half a bee.'  
>>> print_twice(michael)  
Eric, the half a bee.  
Eric, the half a bee.
```

- Il nome della variabile che passiamo come argomento (**michael**) non ha nulla a che fare con il nome del parametro (**bruce**).

# FUNZIONI IN PYTHON

## FRUITFUL FUNCTIONS E VOID FUNCTIONS

- Alcune delle funzioni che possiamo usare, come ad esempio le funzioni matematiche, restituiscono dei risultati. Esse vengono chiamate *fruitful functions* (*funzioni produttive*).
- Altre funzioni, come la **print\_twice** vista prima, vengono chiamate *void functions* (*funzioni vuote*) poiché eseguono un'azione ma non restituiscono alcun valore.



# FUNZIONI IN PYTHON

## FRUITFUL FUNCTIONS E VOID FUNCTIONS

- Quando richiamiamo una *fruitful function*, quasi sempre avremo bisogno di utilizzarne il risultato. Per esempio, potremmo assegnarlo ad una variabile o usarlo come parte di un'espressione:

```
x = math.cos(radians)
golden = (math.sqrt(5) + 1) / 2
```

- Quando si chiama una funzione in modalità interattiva, Python ne visualizza subito il risultato:

```
>>> math.sqrt(5)
2.23606797749979
```

- Quando una *fruitful function* si usa in uno script, occorre memorizzare il risultato in una variabile.

# FUNZIONI IN PYTHON

## FRUITFUL FUNCTIONS E VOID FUNCTIONS

- Le void functions possono visualizzare qualcosa sullo schermo o avere qualche altro effetto, ma non hanno alcun valore di ritorno.
- Se si tenta di assegnare il risultato ad una variabile si otterrà un valore speciale chiamato “None”:

```
>>> result = print_twice('Bing')
```

```
Bing
```

```
Bing
```

```
>>> print(result)
```

```
None
```

- Il valore **None** non è uguale alla stringa “None”. E' un valore speciale che ha un suo proprio tipo:

```
>>> print(type(None))
```

```
<class 'NoneType'>
```

# FUNZIONI IN PYTHON

## FRUITFUL FUNCTIONS E VOID FUNCTIONS

- Mediante l'istruzione **return** inserita nella funzione si ha la possibilità di ottenere il risultato. Ad esempio:

```
def addtwo(a,b):  
    added = a + b  
    return added
```

```
x = addtwo(3, 5)  
print(x)
```

- Quando questo script viene eseguito, l'istruzione **print** visualizza il risultato "8" perché la funzione **addtwo** è stata chiamata passandole gli argomenti 3 e 5.

# ITERAZIONE

## AGGIORNAMENTO DI VARIABILI

- Uno schema comune nelle istruzioni di assegnazione è l'aggiornamento di una variabile in cui il nuovo valore della variabile dipende da quello vecchio. Ad es.:

```
x = x + 1
```

- Se si prova ad aggiornare una variabile che non esiste, si ottiene un messaggio di errore, perché Python verifica il lato destro dell'istruzione prima di assegnare un valore a x:

```
>>> x = x + 1
```

```
NameError: name 'x' is not defined
```

- Prima di poter aggiornare una variabile, occorre *inizializzarla*, solitamente con una semplice assegnazione:

```
>>> x = 0
```

```
>>> x = x + 1
```

# ITERAZIONE

## ISTRUZIONE “WHILE”

- Le iterazioni sono molto comuni, e Python fornisce diverse funzionalità per renderle più semplici. Una di queste è l'istruzione **while**:

```
n = 10
while n > 0 :
    print(n)
    n = n - 1
print('Decollo!')
```

- Il blocco del ciclo dovrebbe cambiare il valore di una o più variabili in modo da far diventare infine falsa la condizione e far sì che il ciclo termini. Chiamiamo variabile di iterazione la variabile che cambia ogni volta che il ciclo viene eseguito e controlla quando il ciclo terminerà.

# ITERAZIONE

## CICLI INFINITI E BREAK

- A volte non si capisce che è il momento di terminare un ciclo fino a quando non si arriva a metà del blocco. In questi casi si può scrivere di proposito un ciclo infinito e poi usare l'istruzione **break** per uscire dal loop.
- Il ciclo seguente è un ciclo infinito perché l'espressione logica nell'istruzione **while** è semplicemente la costante logica **True**:

```
n = 10
while True:
    print(n, end=' ')
    n = n - 1
print('Done!')
```

# ITERAZIONE

## CICLI INFINITI E BREAK

- L'esecuzione del suddetto codice è chiaramente un errore. Possiamo tuttavia usare questo schema per costruire cicli utili, a patto di aggiungere con cura del codice al blocco del ciclo per uscirne esplicitamente usando **break** quando abbiamo raggiunto la condizione di uscita.
- Ad esempio, supponiamo di voler ricevere dati dall'utente fino a quando non digita **done**. Potremmo scrivere:

```
while True:
    line = input('> ')
    if line == 'done':
        break
    print(line)
print('Done!')
```

# ITERAZIONE

## FERMARE LE ITERAZIONI CON “CONTINUE”

- L'istruzione **continue** consente di saltare alla successiva iterazione senza terminare il corpo del ciclo per l'iterazione corrente. Ad esempio:

```
while True:
    line = input('> ')
    if line[0] == '#':
        continue
    if line == 'done':
        break
    print(line)
print('Done!')
```

- Vengono visualizzati tutti gli input, tranne quelli che cominciano con il segno cancelletto.



# ITERAZIONE

## CICLI DEFINITI CON L'USO DI "FOR"

- A volte vogliamo eseguire il ciclo su un *insieme* di oggetti come una lista di parole, le righe di un file o una lista di numeri. In questi casi possiamo costruire un ciclo *definito* usando l'istruzione **for**. Ad esempio, il seguente ciclo:

```
friends = ['Joseph', 'Glenn', 'Sally']  
for friend in friends:  
    print('Happy New Year:', friend)  
print('Done!')
```

stampa il seguente messaggio:

```
Happy New Year: Joseph  
Happy New Year: Glenn  
Happy New Year: Sally  
Done!
```

# ITERAZIONE

## CICLI PER CONTARE E SOMMARE

- Ad esempio, per contare il numero di elementi in una lista, useremo il seguente ciclo **for**:

```
count = 0
for itervar in [3, 41, 12, 9, 74, 15]:
    count = count + 1
print('Count: ', count)
```

- Il seguente ciclo calcola la somma di un insieme di numeri:

```
total = 0
for itervar in [3, 41, 12, 9, 74, 15]:
    total = total + itervar
print('Total: ', total)
```

# ITERAZIONE

## CICLI DI MASSIMO E DI MINIMO

- Per trovare il valore più grande in una lista o una sequenza possiamo usare il seguente frammento di programma:

```
largest = None
print('Before: ', largest)
for itervar in [3, 41, 12, 9, 74, 15]:
    if largest is None or itervar > largest :
        largest = itervar
    print('Loop:', itervar, largest)
print('Largest: ', largest)
```

# ITERAZIONE

## CICLI DI MASSIMO E DI MINIMO

- Per calcolare il valore più piccolo il codice è molto simile, con una piccola modifica:

```
smallest = None
print('Before: ', smallest)
for itervar in [3, 41, 12, 9, 74, 15]:
    if smallest is None or itervar < smallest :
        smallest = itervar
    print('Loop:', itervar, smallest)
print('Smallest: ', smallest)
```

# STRINGHE

## DEFINIZIONE

- Una stringa è una sequenza di caratteri. E' possibile accedere a ciascuno dei caratteri componenti la stringa mediante l'operatore "bracket":

```
>>> fruit = 'banana'
>>> letter = fruit[1]
```

- La seconda istruzione estrae il carattere nella posizione 1 (si estrae il carattere 'a', ossia il 2° carattere della stringa - vedi figura) dalla variabile **fruit** e lo assegna alla variabile **letter**.

b	a	n	a	n	a
[0]	[1]	[2]	[3]	[4]	[5]

# STRINGHE

## LUNGHEZZA DI UNA STRINGA

- La funzione **len** è una funzione built-in che restituisce il numero di caratteri di una stringa:

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

- Per ottenere l'ultimo carattere della stringa dobbiamo sottrarre 1 alla sua lunghezza:

```
>>> length = len(fruit)
>>> last = fruit[length - 1]
>>> print(last)
a
```

# STRINGHE

## SCANSIONE DI UNA STRINGA

- Per scandire una stringa, carattere per carattere, possiamo usare un ciclo **while**:

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

- Un altro modo per scandire una stringa fa uso del ciclo **for**:

```
for char in fruit:
    print(char)
```

# STRINGHE

## STRING SLICES

- Un segmento di una stringa è chiamato *sottostringa* (*slice*). La selezione di una sottostringa è simile a quella di un carattere:

```
>>> s = 'Monty Python'
```

```
>>> print(s[0:5])
```

```
Monty
```

```
>>> print(s[6:12])
```

```
Python
```

- Se si omette il primo indice, la sottostringa parte dall'inizio della stringa. Se si omette il secondo indice, la sottostringa arriva alla fine della stringa:

```
>>> s[:3]
```

```
>>> 'Mon'
```

```
>>> s[6:]
```

```
>>> 'Python'
```



# STRINGHE

## NON SONO MODIFICABILI

- Non è possibile assegnare un valore a una stringa già definita:

```
>>> greeting = 'Hello World!'
>>> greeting[0] = 'J'
TypeError: 'str' object does not support item assignment
```

- E' possibile però creare una nuova stringa come variante dell'originale (mediante l'operatore di *concatenazione* +):

```
>>> greeting = 'Hello World!'
>>> new_greeting = 'J' + greeting[1:]
>>> print(new_greeting)
Jello World!
```

# STRINGHE

## CICLI E CONTEGGI

- Il seguente frammento di codice conta il numero di volte che la lettera **a** compare nella stringa:

```
word = 'banana'
count = 0
for letter in word:
    if letter == 'a':
        count = count + 1
print(count)
```

- La variabile contatore **count** è inizializzata a zero e poi incrementata ogni volta che una **a** è trovata. Quando si esce dal ciclo, **count** contiene il risultato.

# STRINGHE

## L'OPERATORE "IN"

- La parola **in** è un operatore booleano che considera due stringhe e restituisce **True** se la prima stringa è una sottostringa della seconda:

```
>>> 'a' in 'banana'
```

```
True
```

```
>>> 'seed' in 'banana'
```

```
False
```

# STRINGHE

## CONFRONTO TRA STRINGHE

- Gli operatori di confronto possono lavorare su stringhe. Ad esempio, per vedere se due stringhe sono uguali:

```
if word == 'banana':  
    print('All right, bananas.')
```

- Altre operazioni di confronto possono essere utili per porre le parole in ordine alfabetico (in Python le maiuscole vengono prima delle minuscole):

```
if word < 'banana':  
    print('Your word, ' + word + ', comes before banana.')
```

```
elif word > 'banana':  
    print('Your word, ' + word + ', comes after banana.')
```

```
else:  
    print('All right, bananas.')
```

# STRINGHE

## “METODI”

- Le stringhe sono un esempio di *oggetti* in Python.
- Un oggetto contiene sia i dati (la stringa vera e propria) sia i *metodi*, che essenzialmente sono funzioni incorporate nell'oggetto e sono disponibili per qualsiasi istanza dell'oggetto.
- Tramite due funzioni si ha la possibilità di conoscere la tipologia di un oggetto ed i metodi ad esso associati: la funzione **type** mostra il tipo di oggetto e la funzione **dir** mostra i metodi disponibili.

# STRINGHE

## “METODI”

- La chiamata di un metodo è detta *invocazione*.
- Ad esempio, esiste un metodo per le stringhe chiamato **find** che cerca l'eventuale posizione di una stringa o segmento all'interno di un'altra:

```
>>> word = 'banana'
>>> index = word.find('a')
>>> print(index)
1
>>> word.find('na')
2
```

- Questa funzione può ricevere, come secondo argomento, l'indice da cui dovrebbe iniziare la ricerca:

```
>>> word.find('na', 3)
4
```

# STRINGHE

## “METODI”

- Un'altra operazione comune è il rimuovere lo spazio bianco (spazi, tabulazioni o newline) dall'inizio e dalla fine di una stringa usando il metodo **strip**:

```
>>> line = ' Here we go '  
>>> line.strip()  
'Here we go'
```

- Alcuni metodi, come **startswith**, restituiscono valori booleani:

```
>>> line = 'Have a nice day'  
>>> line.startswith('Have')  
True  
>>> line.startswith('h')  
False
```

# STRINGHE

## “METODI”

- Il metodo **capitalize** consente di impostare a maiuscolo la lettera all’inizio della stringa:

```
>>> t = 'questo è un oggetto stringa'
>>> t.capitalize()
'Questo è un oggetto stringa'
```

- Il metodo **upper** imposta a maiuscolo tutti i caratteri:

```
>>> t.upper()
'QUESTO È UN OGGETTO STRINGA'
```



## STRINGHE

### OPERATORE “FORMAT”

- L'operatore % ci consente di costruire stringhe, sostituendo parti di stringhe con dati memorizzati in variabili.
- Il primo operando è il *format string*, che contiene una o più *format sequences* che specificano come deve essere formattato il secondo operando. Il risultato è una stringa.

Esempio:

```
>>> camels = 42
>>> '%d' % camels
'42'
```

Il risultato è la stringa '42'.

## STRINGHE

### OPERATORE “FORMAT”

- Una sequenza di format può comparire in qualsiasi punto nella stringa, in modo da poter inserire un valore in una frase:

```
>>> camels = 42
>>> 'I have spotted %d camels.' % camels
'I have spotted 42 camels.'
```

- Se in una stringa compare più di una sequenza di format, il secondo argomento deve essere una tupla. Ogni sequenza di format è confrontata con un elemento della tupla, in ordine:

```
>>> 'In %d years I have spotted %g %s.' % (3, 0.1, 'camels')
'In 3 years I have spotted 0.1 camels.'
```

In tale esempio abbiamo usato “%d” per formattare un intero, “%g” per formattare un floating-point, e “%s” per formattare una stringa.

# STRINGHE

## OPERATORE “FORMAT”

- Il numero di elementi nella tupla devono corrispondere in numero, ordine e tipo con le sequenze di format nella stringa.

Esempio:

```
>>> '%d %d %d' % (1, 2)
```

```
TypeError: not enough arguments for format string
```

```
>>> '%d' % 'dollars'
```

```
TypeError: '%d' format: a number is required, not str
```

Nel primo caso non ci sono argomenti a sufficienza. Nel secondo l'elemento è di un tipo errato.

# STRINGHE

## NUOVO OPERATORE “FORMAT”

- Una nuova modalità di formattazione messa a disposizione del linguaggio Python è la seguente:

```
>>> 'this is an integer {:d}' .format(15)
'this is an integer 15'
```

```
>>> 'this is a float {:f}' .format(15.3456)
'this is a float 15.345600'
```

```
>>> 'this is a string {:s}' .format('Python')
'this is a string Python'
```

# LISTE

## DEFINIZIONE

- Una *lista* è una sequenza di caratteri (come per le stringhe). Mentre in una stringa i valori possono essere solo caratteri, in una lista essi possono essere di qualsiasi tipo. I valori in una lista sono chiamati *elementi* (*items*) della lista.

Esempio:

```
[10, 20, 30, 40]
```

```
['pippo', 'pluto', 'paperino']
```

Nel primo caso abbiamo una lista di quattro interi. Nel secondo una lista di tre stringhe.

# LISTE

## DEFINIZIONE

- Gli elementi di una lista non devono essere necessariamente tutti dello stesso tipo. La seguente lista contiene una stringa, un float, un intero e un'altra lista:

```
[ 'spam', 2.0, 5, [10, 20] ]
```

- Una lista all'interno di un'altra lista è detta "annidata" (**nested**).
- Una lista che non contiene elementi è chiamata lista vuota. E' possibile creare una lista vuota mediante parentesi quadre vuote: [].

# LISTE

## DEFINIZIONE

- E' possibile assegnare liste a variabili:

```
>>> formaggi = ['Asiago', 'Caciocavallo', 'Gorgonzola']
>>> numeri = [17, 123]
>>> vuota = []
>>> print(formaggi, numeri, vuota)
['Asiago', 'Caciocavallo', 'Gorgonzola'] [17, 123] []
```

# LISTE

## SONO MODIFICABILI

- La sintassi per accedere agli elementi di una lista è lo stesso di quello che si usa per accedere agli elementi di una stringa (operatore “bracket”):

```
>>> print(formaggi[0])
```

```
Asiago
```

- A differenza delle stringhe, le liste sono modificabili aggiornando un item o variando l'ordine degli elementi:

```
>>> numeri = [17, 123]
```

```
>>> numeri[1] = 5
```

```
>>> print(numeri)
```

```
[17, 5]
```



# LISTE

## SONO MODIFICABILI

- Possiamo considerare una lista come un *mapping* tra indici ed elementi.
- Gli indici delle liste funzionano come quelli delle stringhe:
  - Ogni espressione intera può essere usata come indice.
  - Se si prova a leggere o a scrivere un elemento che non esiste, si ottiene un **IndexError**.
  - Se un indice ha un valore negativo, conta all'indietro a partire dalla fine della lista (e.g., [-1] corrisponde all'ultimo elemento della lista, [-2] al penultimo, ecc.).
- L'operatore **in** funziona come per le stringhe:

```
>>> 'Caciocavallo' in formaggi  
True
```

# LISTE

## SCANSIONE DI UNA LISTA

- Un modo per scandire una lista fa uso di un ciclo **for**. La sintassi è la stessa di quella usata per le stringhe:

```
for formaggio in formaggi:  
    print(formaggio)
```

- Se vogliamo scrivere o aggiornare degli elementi possiamo usare gli indici:

```
for i in range(len(numeri)):  
    numeri[i] = numeri[i] * 2
```

# LISTE

## OPERAZIONI

- L'operatore `+` concatena le liste:

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

- Analogamente, l'operatore `*` replica una lista un certo numero di volte:

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

# LISTE

## LIST SLICES

- L'operatore *slice* funziona anche per le liste:

```
>>> t = ['a', 'b', 'c', 'd', 'e', 'f']
>>> t[1:3]
['b', 'c']
>>> t[:4]
['a', 'b', 'c', 'd']
>>> t[3:]
['d', 'e', 'f']
```

- Se si omettono entrambi gli indici, si ottiene una copia dell'intera lista:

```
>>> t[:]
['a', 'b', 'c', 'd', 'e', 'f']
```

# LISTE

## LIST SLICES

- L'operatore slice messo a sinistra di un assegnamento consente di aggiornare più elementi:

```
>>> t = ['a', 'b', 'c', 'd', 'e', 'f']
>>> t[1:3] = ['x', 'y']
>>> print(t)
['a', 'x', 'y', 'd', 'e', 'f']
```

# LISTE

## METODI

- Python fornisce vari *metodi* per operare sulle liste. Per esempio, **append** aggiunge un nuovo elemento alla fine della lista:

```
>>> t = ['a', 'b', 'c']
>>> t.append('d')
>>> print(t)
['a', 'b', 'c', 'd']
```

- extend** prende una lista come argomento e fa l'append di tutti gli elementi:

```
>>> t1 = ['a', 'b', 'c']
>>> t2 = ['d', 'e']
>>> t1.extend(t2)
>>> print(t1)
['a', 'b', 'c', 'd', 'e']
```

# LISTE

## METODI

- **sort** ordina gli elementi di una lista dal minore al maggiore:

```
>>> t = ['d', 'c', 'e', 'b', 'a']
>>> t.sort()
>>> print(t)
['a', 'b', 'c', 'd', 'e']
```

- Molti metodi per le liste sono “void”; modificano la lista e restituiscono **None** (dobbiamo dunque prestare attenzione a istruzioni come questa: **t = t.sort()**, che ci farebbe perdere gli elementi della lista)

# LISTE

## ESTRAZIONE DI ELEMENTI

- Ci sono vari modi per cancellare elementi di una lista. Se si conosce l'indice dell'elemento interessato, possiamo usare **pop**:

```
>>> t = ['a', 'b', 'c']
>>> x = t.pop(1)
>>> print(t)
['a', 'c']
>>> print(x)
b
```

- **pop** modifica la lista e restituisce l'elemento rimosso. Se non si specifica l'indice, elimina e restituisce l'ultimo elemento.



# LISTE

## CANCELLAZIONE DI ELEMENTI

- Se non abbiamo bisogno dell'elemento rimosso, possiamo usare l'operatore **del**:

```
>>> t = ['a', 'b', 'c']
>>> del t[1]
>>> print(t)
['a', 'c']
```

- Se conosciamo l'elemento da rimuovere ma non conosciamo il suo indice, possiamo usare **remove**:

```
>>> t = ['a', 'b', 'c']
>>> t.remove('b')
>>> print(t)
['a', 'c']
```

Il valore restituito da **remove** è **None**.

# LISTE

## CANCELLAZIONE DI ELEMENTI

- Per rimuovere più di un elemento, possiamo utilizzare l'operatore **del** con uno *slice index*:

```
>>> t = ['a', 'b', 'c', 'd', 'e', 'f']
>>> del t[1:5]
>>> print(t)
['a', 'f']
```

- Come sempre, lo *slice* seleziona tutti gli elementi fino al secondo indice, senza includere quest'ultimo.

# LISTE

## FUNZIONI BUILT-IN

- Ci sono varie funzioni built-in che possono essere usate per elaborazioni sulle liste:

```
>>> nums = [3, 41, 12, 9, 74, 15]
```

```
>>> print(len(nums))
```

```
6
```

```
>>> print(max(nums))
```

```
74
```

```
>>> print(min(nums))
```

```
3
```

```
>>> print(sum(nums))
```

```
154
```

```
>>> print(sum(nums)//len(nums))
```

```
25
```

# LISTE

## FUNZIONI BUILT-IN

- La funzione **sum()** funziona solo quando gli elementi della lista sono numeri. Le altre funzioni (**max()**, **len()**, ecc.) funzionano anche con liste di stringhe e altri tipi di dati che possono essere confrontati tra di loro.
- A titolo di esempio, scriviamo un programma che calcola la media di una lista di numeri.

# LISTE

## FUNZIONI BUILT-IN

- Rivediamo la versione del programma che non fa uso di una lista:

```
total = 0
count = 0
while (True):
    inp = input('Enter a number: ')
    if inp == 'done': break
    value = float(inp)
    total = total + value
    count = count + 1
average = total/count
print('Average: ', average)
```

# LISTE

## FUNZIONI BUILT-IN

- Utilizzando una lista, potremmo semplicemente ricordare ciascun numero inserito e usare delle funzioni built-in per calcolarne la somma e la media:

```
numlist = list()
while (True):
    inp = input('Enter a number: ')
    if inp == 'done': break
    value = float(inp)
    numlist.append(value)
average = sum(numlist)/len(numlist)
print('Average: ', average)
```

Il programma crea una lista vuota prima del ciclo while, e fa l'append nella lista di ciascun numero inserito. Alla fine calcoliamo la somma degli elementi della lista e la dividiamo per la sua lunghezza.

# LISTE

## LISTE E STRINGHE

- E' possibile convertire una stringa in una lista usando la funzione **list**:

```
>>> s = 'spam'
>>> t = list(s)
>>> print(t)
['s', 'p', 'a', 'm']
```

- La funzione **list** suddivide la stringa in caratteri singoli. Se vogliamo suddividere la stringa in parole singole possiamo usare il metodo **split**:

```
>>> s = 'Introduzione al Python'
>>> t = s.split()
>>> print(t)
['Introduzione', 'al', 'Python']
>>> print(t[2])
'Python'
```

# LISTE

## LISTE E STRINGHE

- E' possibile utilizzare la funzione **split** con un argomento opzionale chiamato *delimiter*, che specifica quali caratteri usare come delimitatori. Nel seguente esempio usiamo il carattere `'-'`:

```
>>> s = 'spam-spam-spam'
>>> delimiter = '-'
>>> s.split(delimiter)
['spam', 'spam', 'spam']
```



# LISTE

## LISTE E STRINGHE

- Una funzione inversa di **split** è la **join** che prende una lista di stringhe e concatena gli elementi. **join** è un metodo per le stringhe, quindi occorre richiamarlo per mezzo del delimitatore e passare la lista come argomento:

```
>>> t = ['Introduzione', 'al', 'Python']
>>> delimiter = ' '
>>> delimiter.join(t)
'Introduzione al Python'
```

- In questo caso il delimitatore è uno spazio, perciò **join** ne aggiunge uno tra le varie parole. Se avessimo bisogno di concatenare delle stringhe senza spazi, potremmo usare come delimitatore la stringa vuota "".

# LISTE

## OGGETTI E VALORI

- Se eseguiamo le seguenti istruzioni di assegnazione:

**a** = 'banana'

**b** = 'banana'

sappiamo che **a** e **b** si riferiscono a una stringa, ma non sappiamo se si riferiscono alla *stessa* stringa.

Ci sono due possibili stati:

**a** → 'banana'

**b** → 'banana'

**a** → 'banana'  
**b** → 'banana'

In un caso, **a** e **b** si riferiscono a due differenti oggetti che hanno lo stesso valore. Nell'altro caso, si riferiscono allo stesso oggetto.

# LISTE

## OGGETTI E VALORI

- Per verificare se due variabili si riferiscono allo stesso oggetto, possiamo usare l'operatore **is** come segue:

```
>>> a = 'banana'
>>> b = 'banana'
>>> a is b
```

**True**

In questo esempio, Python ha creato un solo oggetto stringa, ed entrambe le variabili **a** e **b** si riferiscono ad esso.

- Se invece creiamo due liste, otteniamo due oggetti:

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

**False**

## LISTE

### OGGETTI E VALORI

- In quest'ultimo caso diciamo che le due liste sono *equivalenti*, poiché hanno gli stessi elementi, ma non *identiche*, poiché non costituiscono lo stesso oggetto.
- Se due oggetti sono identici, essi sono anche equivalenti. Se invece sono equivalenti, essi non sono necessariamente identici.
- Fino ad ora abbiamo usato i termini “oggetto” e “valore” in modo intercambiabile, ma è più preciso dire che un oggetto ha un valore. Se ad esempio eseguiamo la seguente istruzione:

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

la variabile **a** si riferisce a un oggetto lista il cui valore è una particolare sequenza di elementi. Se un'altra lista ha gli stessi elementi, diciamo che essa ha lo stesso valore.

## LISTE

### ALIASING

- Se la variabile **a** si riferisce a un oggetto e assegniamo **b = a**, entrambe le variabili si riferiscono allo stesso oggetto:

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

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

```
>>> b is a
```

```
True
```

- L'associazione di una variabile a un oggetto è chiamata *reference* (*riferimento*). Nell'esempio ci sono due riferimenti allo stesso oggetto.
- Un oggetto con più di un riferimento ha più di un nome, e possiamo dire che l'oggetto è *aliased* (ha un *alias*).

## LISTE

### ALIASING

- Se l'oggetto aliased è mutabile, le modifiche fatte in uno degli alias risulteranno anche nell'altro:

```
>>> b[0] = 17
```

```
>>> print(a)
```

```
[17, 2, 3]
```

- Sebbene questo comportamento possa essere utile, può essere comunque fonte di errori. E' consigliabile quindi evitare l'aliasing quando lavoriamo con oggetti mutabili.
- Per oggetti immutabili come le stringhe, l'aliasing non rappresenta un problema. In questo esempio:  

```
a = 'banana'
```

```
b = 'banana'
```

in genere non fa molta differenza se **a** e **b** si riferiscono o no alla stessa stringa.

## LISTE ARGOMENTI

- Quando si passa una lista a una funzione, la funzione riceve il riferimento alla lista. Se la funzione modifica un parametro della lista, il “chiamante” vede la modifica.
- Per esempio, la seguente funzione **delete\_head** rimuove il primo elemento della lista:

```
def delete_head(t):  
    del t[0]
```

Ecco come può essere usata:

```
>>> letters = ['a', 'b', 'c']  
>>> delete_head(letters)  
>>> print(letters)  
['b', 'c']
```

Il parametro **t** e la variabile **letters** sono *aliased* per lo stesso oggetto.

# LISTE

## ARGOMENTI

- E' importante distinguere tra operazioni che modificano liste e operazioni che creano nuove liste.
- Per esempio, il metodo **append** modifica una lista, mentre l'operatore **+** crea una nuova lista:

```
>>> t1 = [1, 2]
>>> t2 = t1.append(3)
>>> print(t1)
[1, 2, 3]
>>> print(t2)
None
```

```
>>> t1 = [1, 2]
>>> t3 = t1 + [3]
>>> print(t3)
[1, 2, 3]
>>> t1 is t3
False
```



# LISTE

## ARGOMENTI

- La differenza che abbiamo evidenziato diventa importante quando scriviamo funzioni che debbano modificare liste. Ad esempio, la funzione che segue NON cancella il primo elemento della lista che gli passiamo:

```
def bad_delete_head(t):  
    t = t[1:]           # errore!
```

L'operatore *slice* crea una nuova lista e l'istruzione di assegnazione fa in modo che **t** si riferisca ad essa, ma ciò non ha nessun effetto sulla lista che era stata passata alla funzione.

## LISTE ARGOMENTI

- Un'alternativa è quella di scrivere una funzione che crea e restituisce una nuova lista. Ad esempio, la seguente funzione **tail** restituisce la lista senza il primo elemento:

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

Tale funzione lascia la lista originale inalterata. Ecco come può essere usata:

```
>>> letters = ['a', 'b', 'c']  
>>> rest = tail(letters)  
>>> print(rest)  
['b', 'c']
```

# LISTE

## LIST COMPREHENSION

- In Python c'è uno speciale costrutto che consente di accedere via via a tutti gli elementi di una lista, effettuare la stessa operazione su ciascuno di essi, e memorizzare i nuovi elementi ottenuti in un'altra lista. Ad esempio:

```
>>> lista1 = [1, 2, 3, 4, 5]
>>> lista2 = [e * 2 for e in lista1]
>>> print(lista2)
[2, 4, 6, 8, 10]
```

# LISTE

## ATTRAVERSAMENTO SIMULTANEO DI PIÙ LISTE

- Accade di frequente che due o più liste debbano essere attraversate simultaneamente. Python offre una speciale e comoda sintassi che può essere descritta come segue:

```
for e1, e2, e3, ... in zip(lista1, lista2, lista3, ...):  
    # lavora con l'elemento e1 della lista1, e2 della lista2,  
    # e3 della lista3, ecc.
```

Ad esempio, con il seguente frammento di codice:

```
lista1 = [3, 2, 5, 7, 10]  
lista2 = [5, 4, 2, 9, 5, 3, 8]  
l = []  
for a, b in zip(lista1, lista2):  
    l.append(a + b)
```

si ottiene la lista: `l = [8, 6, 7, 16, 15]`

# DIZIONARI

## DEFINIZIONE

- Un *dizionario* è simile a una lista, ma di tipo più generale. In una lista gli indici che individuano le varie posizioni devono essere degli interi. In un dizionario gli indici possono essere di vari tipi.
- Possiamo pensare a un dizionario come a un mapping tra un insieme di indici, chiamati chiavi (*keys*), e un insieme di valori.
- Ogni chiave individua un valore. L'associazione tra chiavi e valori è chiamata *key-value pair* o a volte *item*.

# DIZIONARI

## ESEMPIO DI APPLICAZIONE

- Come esempio di applicazione, costruiamo un dizionario che ci consenta di passare da parole in inglese a parole in spagnolo. Le chiavi e i valori sono dunque delle stringhe.
- La funzione built-in **dict** crea un dizionario senza item:

```
>>> eng2sp = dict()  
>>> print(eng2sp)  
{}
```

Le parentesi graffe rappresentano un dizionario vuoto.

# DIZIONARI

## ESEMPIO DI APPLICAZIONE

- Per aggiungere elementi al dizionario si possono utilizzare le parentesi quadre:

```
>>> eng2sp[ 'one' ] = 'uno'
```

Questa istruzione associa la chiave **'one'** al valore **'uno'**. Se stampiamo di nuovo il dizionario, otteniamo la coppia key-value come segue:

```
>>> print(eng2sp)
{ 'one' : 'uno' }
```

# DIZIONARI

## ESEMPIO DI APPLICAZIONE

- Il formato di output ottenuto è anche un formato di input. Per esempio, possiamo creare un nuovo dizionario con tre item come segue:

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

L'ordine delle coppie key-value non è lo stesso. Facendo eseguire le istruzioni su un altro computer possiamo ottenere un risultato differente. In generale, l'ordine degli item in un dizionario non è prevedibile.



# DIZIONARI

## ESEMPIO DI APPLICAZIONE

- L'ordine degli item non è però un problema perché gli elementi di un dizionario non sono mai indicizzati mediante indici interi. In un dizionario si accede ai valori attraverso le chiavi:

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

La chiave **'two'** corrisponde al valore **'dos'**, pertanto l'ordine degli item non conta.

- Se una chiave non è presente nel dizionario, si ha una eccezione:

```
>>> print(eng2sp['four'])  
KeyError: 'four'
```

# DIZIONARI

## FUNZIONE “LEN” E OPERATORE “IN”

- Per il calcolo del numero degli item in un dizionario possiamo usare la funzione **len**:

```
>>> len(eng2sp)
3
```

- L'operatore **in** funziona anche per i dizionari: segnala se è presente una chiave nel dizionario (se è presente come “valore” non è rilevante):

```
>>> 'one' in eng2sp
True
>>> 'uno' in eng2sp
False
```

## DIZIONARI

### IL METODO “VALUES”

- Per vedere se è presente un valore nel dizionario, possiamo usare il metodo **values**, che restituisce i valori in una lista, e poi usare l'operatore **in**:

```
>>> vals = list(eng2sp.values())  
>>> 'uno' in vals  
True
```

- L'operatore **in** usa differenti algoritmi per le liste e per i dizionari. Per le liste usa una *linear search*. Per i dizionari una *hash table*.

## DIZIONARI COME INSIEME DI CONTATORI

- Supponiamo di voler contare le occorrenze di ciascun carattere che compare in una stringa. Ci sono varie possibilità:
  - Creare 26 variabili, una per ciascuna lettera dell'alfabeto. Si scandisce la stringa e, per ogni carattere incontrato, si incrementa il contatore corrispondente ( ad es. usando degli **if** in cascata).
  - Creare una lista con 26 elementi. Convertire poi ciascun carattere in un numero (usando la funzione built-in **ord**); usare tale numero come indice per la lista e incrementare l'appropriato contatore.
  - Creare un dizionario con i caratteri come chiavi e i corrispondenti valori come contatori. La prima volta che si incontra un carattere, si aggiunge un item al dizionario. Successivamente si incrementa il valore associato ad ogni carattere incontrato.

## DIZIONARI COME INSIEME DI CONTATORI

- Un vantaggio della soluzione che fa uso di un dizionario è quello di non dover sapere a priori quali lettere compaiono nella stringa.

Ecco un esempio di codice:

```
word = 'brontosaurus'
d = dict()
for c in word:
    if c not in d:
        d[c] = 1
    else:
        d[c] = d[c] + 1
print(d)
```

che dà luogo a questo output:

```
{'a': 1, 'b': 1, 'o': 2, 'n': 1, 's': 2, 'r': 2, 'u': 2, 't': 1}
```

# DIZIONARI

## IL METODO “GET”

- I dizionari hanno a disposizione un metodo chiamato **get** che, a fronte di una chiave, restituisce il corrispondente valore. Se la chiave non è presente nel dizionario, restituisce un valore di default.

Esempio:

```
>>> counts = {'chuck': 1, 'annie': 42, 'jan': 100}
>>> print(counts.get('jan', 0))
100
>>> print(counts.get('tim', 0))
0
```

# DIZIONARI

## IL METODO “GET”

- Possiamo utilizzare **get** per riscrivere il codice precedente, che calcola le occorrenze, in modo più conciso:

```
word = 'brontosaurus'
d = dict()
for c in word:
    d[c] = d.get(c, 0) + 1
print(d)
```

L'uso del metodo **get** ci consente di eliminare l'istruzione **if**.

# TUPLE

## DEFINIZIONE

- Una *tupla* è una sequenza di valori, simile ad una lista.
- I valori memorizzati in una tupla possono essere di qualsiasi tipo, e sono indicizzati da interi.
- Una importante differenza è che le tuple sono *immutabili*.
- Sulle tuple è possibile fare operazioni di *comparazione* e *hash*; quindi è possibile ordinare liste di tuple ed usare tuple come key values nei dizionari Python.



# TUPLE

## DEFINIZIONE

- Sintatticamente, una tupla è una lista di valori separati da virgole:

```
>>> t = 'a', 'b', 'c', 'd', 'e'
```

- Sebbene non sia necessario, è molto comune includere le tuple in parentesi per poterle identificare comodamente quando esaminiamo del codice Python:

```
>>> t = ('a', 'b', 'c', 'd', 'e')
```

# TUPLE

## DEFINIZIONE

- Per creare una tupla con un singolo elemento occorre mettere una virgola alla fine:

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

- Senza la virgola, Python tratterebbe ('a') come una espressione con una stringa tra parentesi:

```
>>> t2 = ('a')
>>> type(t2)
<class 'str'>
```

# TUPLE

## DEFINIZIONE

- Un altro modo di costruire una tupla è quello di utilizzare la funzione **tuple**:

```
>>> t = tuple()
>>> print(t)
()
```

- Se l'argomento è una sequenza (stringa, lista o tupla) il risultato della chiamata a **tuple** sarà a sua volta una tupla composta dagli elementi della sequenza:

```
>>> t = tuple('Italia')
>>> print(t)
('I', 't', 'a', 'l', 'i', 'a')
```

# TUPLE

## DEFINIZIONE

- Molti degli operatori per le liste funzionano anche per le tuple.
- L'operatore parentesi quadra permette di indicare la posizione di un elemento:

```
>>> t = ('a', 'b', 'c', 'd', 'e')
>>> print(t[0])
'a'
```

- L'operatore slice permette di indicare un intervallo di elementi:

```
>>> print(t[1:3])
('b', 'c')
```

# TUPLE

## DEFINIZIONE

- Se però si tenta di modificare uno degli elementi della tupla si ottiene un messaggio di errore:

```
>>> t[0] = 'A'
```

```
TypeError: 'tuple' object does not support item assignment
```

- Non è dunque possibile modificare gli elementi di una tupla, ma è possibile sostituire una tupla con un'altra:

```
>>> t = ('A',) + t[1:]
```

```
>>> print(t)
```

```
('A', 'b', 'c', 'd', 'e')
```

# TUPLE

## COMPARAZIONE

- Gli operatori di confronto funzionano con le tuple e con le altre sequenze.
- Python comincia confrontando il primo elemento di ciascuna sequenza. Se sono uguali passa a confrontare l'elemento successivo, e così via finché non ne trova due diversi. Gli elementi successivi non vengono presi in considerazione:

```
>>> (0, 1, 2) < (0, 3, 4)
```

```
True
```

```
>>> (0, 1, 2000000) < (0, 3, 4)
```

```
True
```

# TUPLE COMPARAZIONE

- La funzione **sort** funziona allo stesso modo. Essa ordina cominciando dal primo elemento, ma in caso di pareggio passa al secondo elemento, e così via.
- Questa caratteristica può essere utile in uno schema chiamato **DSU**:
  - **Decorate**: “Decorare” una sequenza costruendo un elenco di tuple con una o più chiavi di ordinamento che precedono gli elementi della sequenza.
  - **Sort**: Ordinare la lista delle tuple usando il **sort** di Python.
  - **Undecorate**: Eliminare la “decorazione” estraendo gli elementi ordinati della sequenza.

# TUPLE

## COMPARAZIONE

- Esempio: ordinare un elenco di parole dalla più lunga alla più corta:

```
txt = 'but soft what light in yonder window breaks'
words = txt.split()
t = list()
for word in words:
    t.append((len(word), word))
t.sort(reverse=True)
res = list()
for length, word in t:
    res.append(word)
print(res)
```

- L'output del programma è il seguente:

```
['yonder' 'window' 'breaks' 'light' 'what' 'soft' 'but' 'in']
```



# TUPLE

## ASSEGNAZIONE

- Una delle caratteristiche sintattiche uniche del linguaggio Python è la possibilità di avere una tupla sul lato sinistro di un'istruzione di assegnazione.
- Ciò consente di assegnare più di una variabile alla volta quando il lato sinistro è una sequenza.

Esempio:

```
>>> m = [ 'have' , 'fun' ]  
>>> x, y = m  
>>> x  
'have'  
>>> y  
'fun'
```

# TUPLE

## ASSEGNAZIONE

- Python traduce *approssimativamente* la sintassi dell'assegnazione della tupla come segue:

```
>>> m = [ 'have' , 'fun' ]
```

```
>>> x = m[0]
```

```
>>> y = m[1]
```

```
>>> x
```

```
'have'
```

```
>>> y
```

```
'fun'
```

# TUPLE

## ASSEGNAZIONE

- Stilisticamente, in genere quando viene utilizzata una tupla sul lato sinistro dell'istruzione di assegnazione non vengono utilizzate le parentesi. In ogni caso la sintassi seguente è altrettanto valida:

```
>>> m = ['have', 'fun']
```

```
>>> (x, y) = m
```

```
>>> x
```

```
'have'
```

```
>>> y
```

```
'fun'
```

## TUPLE ASSEGNAZIONE

- Un'applicazione ingegnosa dell'assegnazione di tuple ci consente di scambiare i valori di due variabili con una singola istruzione:

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

- Entrambi i lati di questa istruzione sono tuple: sul lato sinistro c'è una tupla di variabili, nel lato destro c'è una tupla di espressioni.
- Ogni valore sul lato destro viene assegnato alla rispettiva variabile sul lato sinistro. Tutte le espressioni sul lato destro sono valutate prima di qualsiasi assegnazione.

# TUPLE

## ASSEGNAZIONE

- Il numero di variabili a sinistra e il numero di valori a destra devono essere uguali:

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

```
ValueError: too many values to unpack
```

- Più in generale, il lato destro può contenere un qualsiasi tipo di sequenza (stringa, lista o tupla). Ad esempio, per suddividere un indirizzo email in nome utente e dominio, è possibile scrivere:

```
>>> addr = 'monty@python.org'
```

```
>>> uname, domain = addr.split('@')
```

```
>>> print(uname)
```

```
monty
```

```
>>> print(domain)
```

```
python.org
```

# TUPLE

## DIZIONARI E TUPLE

- I dizionari supportano un metodo chiamato **items** che restituisce un elenco di tuple, in cui ogni tupla è una coppia chiave-valore:

```
>>> d = {'a':10, 'b':1, 'c':22}
>>> t = list(d.items())
>>> print(t)
[('b', 1), ('a', 10), ('c', 22)]
```

- Come possiamo aspettarci da un dizionario, gli elementi non sono in un ordine particolare.

# TUPLE

## DIZIONARI E TUPLE

- Tuttavia, poiché l'elenco di tuple è una lista e le tuple sono comparabili, possiamo ordinare la lista di tuple. Convertire un dizionario in un elenco di tuple è un modo per far sì che sia possibile ordinare il contenuto di un dizionario in base a una chiave:

```
>>> d = {'a':10, 'b':1, 'c':22}
>>> t = list(d.items())
>>> t
[('b', 1), ('a', 10), ('c', 22)]
>>> t.sort()
>>> t
[('a', 10), ('b', 1), ('c', 22)]
```

- Il nuovo elenco viene ordinato secondo un ordine alfabetico crescente del valore della chiave.

# TUPLE

## ASSEGNAZIONE MULTIPLA CON DIZIONARI

- Combinando **items**, assegnazione di tuple e **for** è possibile individuare un modello di codice molto interessante che scorra le chiavi e i valori di un dizionario in un singolo ciclo:

```
for key, val in list(d.items()):  
    print(val, key)
```

- Questo ciclo ha due *variabili di iterazione*, perché **items** restituisce un elenco di tuple e **key, val** è un'assegnazione di tupla che successivamente si ripete nel dizionario attraverso ciascuna delle coppie chiave-valore.



# TUPLE

## ASSEGNAZIONE MULTIPLA CON DIZIONARI

- Per ogni iterazione nel ciclo, sia key che value avanzano alla successiva coppia chiave-valore nel dizionario (sempre in ordine di hash).
- L'output di questo ciclo è:  
  
**10 a**  
**22 c**  
**1 b**
- Di nuovo, l'ordine è basato sul valore dell'hash (cioè, nessun ordine particolare).

# TUPLE

## ASSEGNAZIONE MULTIPLA CON DIZIONARI

- Se combiniamo queste due tecniche, possiamo visualizzare il contenuto di un dizionario ordinato per il *valore* memorizzato in ciascuna coppia chiave-valore.
- Per fare ciò, dobbiamo prima creare una lista di tuple in cui ogni tupla è: **(valore, chiave)**.
- Il metodo **items** ci fornisce un elenco di tuple **(chiave, valore)**, che questa volta vogliamo ordinare per valore e non per chiave.

# TUPLE

## ASSEGNAZIONE MULTIPLA CON DIZIONARI

- Una volta creato l'elenco con le tuple chiave-valore è semplice ordinare l'elenco in ordine inverso e visualizzare il nuovo elenco:

```
>>> d = {'a':10, 'b':1, 'c':22}
>>> l = list()
>>> for key, val in d.items() :
...     l.append((val, key))
...
>>> l
[(10, 'a'), (22, 'c'), (1, 'b')]
>>> l.sort(reverse=True)
>>> l
[(22, 'c'), (10, 'a'), (1, 'b')]
```

- Costruendo attentamente l'elenco di tuple in modo da avere il valore come primo elemento di ciascuna tupla possiamo ordinare l'elenco di tuple in base al valore.

# SETS

## INTRODUZIONE

- I “set objects” sono collezioni non ordinate di altri oggetti, in cui ogni elemento compare una sola volta.
- Ad esempio, con l’istruzione che segue si crea un set **s** a partire da una lista in cui alcuni elementi sono ripetuti:

```
>>> s = set(['u', 'd', 'ud', 'du', 'd', 'du'])  
>>> print(s)  
>>> {'d', 'du', 'u', 'ud'}
```

# SETS

## OPERAZIONI DI BASE

- Vediamo le operazioni di base su insiemi, applicate sul set `s` precedente e su un altro set `t`:

```
>>> t = set(['d', 'dd', 'uu', 'u'])
```

```
>>> s.union(t)
```

```
{'d', 'dd', 'du', 'u', 'ud', 'uu'}
```

```
>>> s.intersection(t)
```

```
{'d', 'u'}
```

```
>>> s.difference(t)
```

```
{'du', 'ud'}
```

```
>>> s.symmetric_difference(t)
```

```
{'dd', 'du', 'ud', 'uu'}
```

# SETS

## ESEMPIO DI APPLICAZIONE

- Un'applicazione dei set può essere quella di evitare duplicati in una lista. Ad esempio:

```
>>> from random import randint
>>> l = [randint(0, 10) for i in range(1000)]
>>> len(l)
1000

>>> l[:20]
[6, 7, 1, 2, 8, 7, 8, 10, 0, 3, 1, 10, 7, 4, 6, 8, 1, 0, 1, 1]

>>> s = set(l)
>>> print(s)
{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10}
```

## RIFERIMENTI

Lubanovic, B. *Introducing Python*, O'Reilly, 2020.

Severance, C. *Python for Everybody - Exploring Data in Python 3*, 2016.

Hilpisch, Y. *Python for Finance*, 2nd edition, O'Reilly, 2019.

Hill, C. *Learning Scientific Programming with Python*, Cambridge University Press, 2020.

Horstmann, C., Necaie, R.D. *Python for Everyone*, John Wiley & Sons, 2019.

Langtangen, H.P. *A Primer on Scientific Programming with Python*, 2nd Edition, Springer, 2020.